高速化推進研究活動報告

第8号

Cyberscience
Center

東北大学サイバーサイエンスセンター

高速化推進研究活動報告 第 8 号

目次

1. 高速化推進研究活動報告の刊行にあたって

サイバーサイエンスセンター長　菅沼拓夫

東北大学サイバーサイエンスセンターは，全国共同利用施設として 1969 年に設置された東北大学大型計算機センターを起源としています．本センターの目的は「最先端かつ世界最大級のコンピュータシステムの導入と利用環境の構築」であり，設立以来，半世紀以上にわたって最新鋭のコンピュータ利用環境を全国の研究者に提供してきました．

計算機の揺籃期には，大型計算機センター，計算機メーカ，利用者が一体となってコンピュータのハードウェア，ソフトウェア，プログラミング言語等が開発されました．また，これら最先端の機器や技術を使いこなすために，上記 3 者が協力して利用環境を整備し，さらに分かりやすいマニュアルの作成やプログラミング相談，講習会等で利用技術等の普及に努めてきました．本センターは，3 者によって高速化を推進することを目的とした研究会を 1997 年 9 月に立ち上げ，それ以降，高性能計算に関する共同研究を精力的に進めてきました．さらには，2014 年に高性能計算技術開発(NEC)共同研究部門を新設し，本センター利用者にとって真に役立つ学術情報基盤の整備・運用・研究開発に取り組んでまいりました．

本センターを取り巻く環境は時代とともに大きく変化しています．2007 年の最先端研究施設共用イノベーション事業による民間企業の利用開始，2010 年の学際大規模情報基盤共同利用・共同研究(JHPCN) 拠点認定，および JHPCN 共同研究課題の実施，2012 年の High Performance Computing Infrastructure (HPCI)への資源提供の開始等を経て，その役割が大きく拡大しました．さらには大規模計算システムの利用目的の多様化も進み，従来からの数値シミュレーションに加えて，大規模データ処理や機械学習等の用途でも当たり前のように使われるようになっています．東北大学青葉山キャンパスの直近に建設された 3GeV 高輝度放射光施設 NanoTerasu が 2024 年度から運用を開始し，そこで生じる大量なデータを蓄積・解析する役割も求められています．常に最新・最先端のシステムを運用してその高度利用技術を利用者に提供し続けるという本センターの使命を果たし続けるために，今後も利用支援や共同研究を精力的かつ継続的に取り組んでまいります．

今回お届けする「高速化推進研究活動報告第 8 号」は，本センターにおける利用支援や共同研究等の活動の成果をまとめたもので，2018 年度から 2022 年度までの成果が収録されています．本報告書を通じて，本センターのこれまでの活動内容についてご理解いただくとともに，皆様のプログラムの高速化の一助になれば幸甚です．

最後に，本センターとの共同研究に積極的に参加し，目覚ましい成果を挙げていただいた本センター利用者各位，日ごろからご支援いただく文部科学省および HPCI コンソーシアムの関係各位，および日本電気株式会社に厚くお礼申し上げます．

2. 高速化推進研究活動報告

スーパーコンピューティング研究部　高橋慧智　滝沢寛之　下村陽一

2.1. はじめに

現在，スーパーコンピュータは多様な科学技術分野・学術分野において，研究・開発を加速する計算基盤としてだけでなく，気象予測，洪水被害予測，津波浸水被害予測など，われわれの安全・安心な暮らしを支える社会基盤としても重要な役割を担っている．その結果，高い演算性能に対する要求は留まることを知らず，これらのニーズに応えるべく，サイバーサイエンスセンターは提供計算資源とそのサービスの質の向上に努めている．一方，近年のスーパーコンピュータは，メニーコア化，メモリ階層の深化，異種プロセッサ搭載の複合型システムの普及など，システムの大規模・複雑化が進み，スーパーコンピュータの性能を引き出すためには，計算機科学の知識がすでに必要不可欠となっている．本センターでは，1997年からスーパーコンピューティング研究部，高性能計算技術開発（NEC）共同研究部門，共同利用支援係，共同研究支援係の計算機科学に関する知識と経験を計算科学者である利用者と共有するべく，臨床学的な視点からプログラムの高速化技術と新しいシミュレーション技術開発に関する共同研究を推進している．これらの共同研究を通して得られた知見を将来のシステム設計に反映させることで，利用者にとって使い勝手の良いシステムの実現に向けた研究開発も行っている．本章では本センターにおける高速化推進研究活動について述べる．

2.2. 大規模科学計算システム

本センターでは，1986年に高性能計算センターとして活動を開始して以来，SX-1（NEC製，0.57GFlop/s）から一貫して，主力計算システムとしてベクトル型スーパーコンピュータを導入し，最先端の学術研究を強力に支援，推進してきた．また，本センターは，全国共同利用型の情報基盤センターとしてだけでは無く，2013年度からはフラグシップシステムを中核とする全国の基盤センター等の計算機資源を連携した革新的ハイパフォーマンス・コンピューティング・インフラ（HPCI）の構成機関として，HPCIシステムの構築と多様なユーザニーズに応える高性能計算環境の整備にも取り組んでいる．

高速化支援活動の詳細説明に先立ち，本センターの大規模科学計算システム AOBA の概要を述べる．図 2.2-1 に大規模科学計算システムを示す．AOBA は AOBA-S，AOBA-A および AOBA-B の 3 つのサブシステムから構成される．AOBA-S（504 ノード，21.05PFlop/s，504TB）は 2023 年 8 月より本格運用されており，NEC 製 SX-Aurora TSUBASA ベクトル型スーパーコンピュータを採用している．AOBA-A および AOBA-B はいずれも 2020 年 10 月に運用開始しており，AOBA-A（72 ノード，1.48PFlop/s，45TB）は AOBA-S と同じく SX-Aurora TSUBASA を採用している． AOBA-B（68 ノード，278.5TFlop/s，17TB）は NEC 製 LX406Rz-2 スカラ型並列コンピュータである．主力システムはその規模が示すとおり AOBA-S であり，主にユーザが自ら開発した大規模シミュレーションコードの実行を担っている．一方，ベクトル型アーキテクチャに適さないアプリケーションや，汎用・商用のアプリケーションの実行には AOBA-B が活用される．AOBA-B には，ポスト処理等のためにスカラ型並列コンピュータと密に連携が必要なアプリケーションのために，小規模なベクトル型スーパーコンピュータである AOBA-A が付随している．

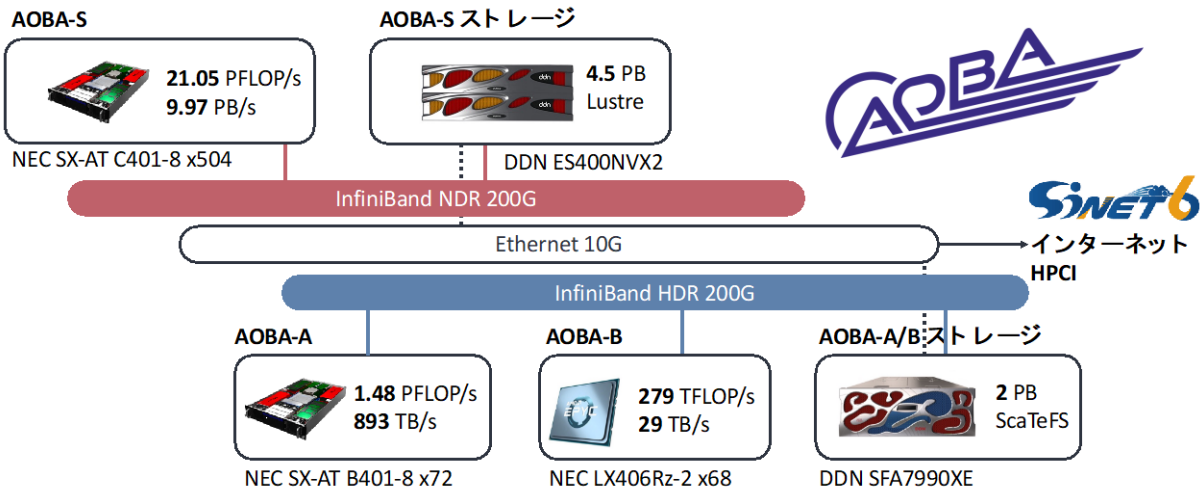

図 2.2-1　本センターの大規模科学計算システム

各サブシステムの概要は次の通りである．AOBA-S は 504 ノードの SX-Aurora TSUBASA システムから構成され，各ノードは 8 基の第 3 世代 NEC 製ベクトルエンジンを搭載している．ベクトルエンジンは PCI Express カードとして実装されたベクトルプロセッサであり，1 基あたり 4.9TFlop/s の理論演算性能および 2.45TB/s のメモリ帯域幅を備える．ホスト側には，64 コアの AMD EPYC 7763 プロセッサおよび 256GB のメモリを備えている．AOBA-S サブシステムの理論演算性能は 21.05PFlop/s，総メモリバンド幅は 9.97PB/s，総メモリ容量は 504TB に達し，大規模シミュレーションを可能にしている．ノード間は InfiniBand NDR インターコネクトによって相互接続しており，同インターコネクトには 4.5PB の Lustre ストレージも接続されている．

AOBA-A は 72 ノードの SX-Aurora TSUBASA システムであり，1 ノードあたり 8 基の第 2 世代ベクトルエンジンを搭載し，ホスト側には 24 コアの AMD EPYC 7402P プロセッサおよび 256GB のメモリを備えている．スカラ型並列コンピュータ AOBA-B は，64 コアの AMD EPYC 7702 プロセッサおよび 256GB のメモリを搭載している．AOBA-A と B は InfiniBand HDR インターコネクトによって結合しており，AOBA-A で得られたシミュレーション結果を AOBA-B で可視化するなど，ベクトルとスカラを連携させたワークフロー処理を実現することが可能である．両サブシステムは 2PB の ScaTeFS ストレージを共有している．

2.3.　高速化推進研究活動

本センターでは1997年より，ユーザアプリケーションの高精度化，大規模化の支援を目的とした共同研究制度を施行している．計算科学を専門とする利用者と計算機科学を専門とするセンター教職員が連携して，アプリケーションの高速化に取り組んでいる．また，本センターでは社会貢献の一環として，サイバーサイエンスセンター共同研究制度の他に，産学連携共同研究に基づく民間利用制度も実施しており，学術分野のみならず産業のイノベーション創出にも貢献してきた．また，本センターは，全国共同利用型の情報基盤センター群と連携して学際大規模情報基盤共同利用・共同研究拠点（JHPCN）を形成し，2010年度にネットワーク型共同利用・共同研究拠点として文部科学大臣の認定を受け，超大規模計算機と超大容量のストレージおよび超大容量ネットワークなどの情報基盤を用いてグランドチャレンジ的な問題について，学際的な共同利用・共同研究を実施している．2013年度からは，フラグシップシステムを

中核とする全国の基盤センター等の計算機資源を連携した革新的ハイパフォーマンス・コンピューティング・インフラ(HPCI)資源提供機関としても活動しており，HPCI採択課題における共同研究を実施している．

図2.3-1に各共同研究の対象領域を示す．サイバーサイエンスセンター共同研究は，研究室レベルから本センターに代表される情報基盤センターのスーパーコンピュータで実行されるシミュレーションコードを対象としており，JHPCN 共同研究はスーパーコンピュータを中心としたシミュレーション規模の研究課題を対象としている．HPCI 公募研究はスーパーコンピュータ「富岳」に代表されるフラグシップシステム，もしくはそれに準ずる規模のシミュレーションコードを取り扱う課題である．

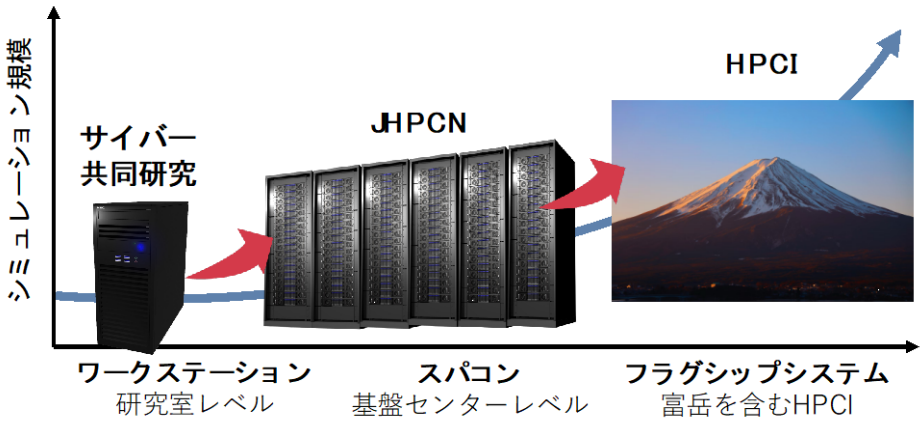

図 2.3-1　共同研究制度とシミュレーション規模

図2.3-2に1999年から本センターで取り組んでいる共同研究数の推移を示す．この図を見ても分かる通り，サイバーサイエンスセンター共同研究は恒常的に年 10 件程度実施されていることに加えて，近年，JHPCN，HPCIを介した共同研究数が増加していることが確認できる．これは，サイバーサイエンスセンター共同研究を通してユーザアプリケーションが高度化・大規模化し，JHPCNあるいはHPCI採択課題へとステップアップしているためであり，われわれの継続的な高速化支援活動が一定の成果を挙げていることがわかる．また，継続的な産学連携に基づく共同研究を実施し，その成果を広く社会に還元している．

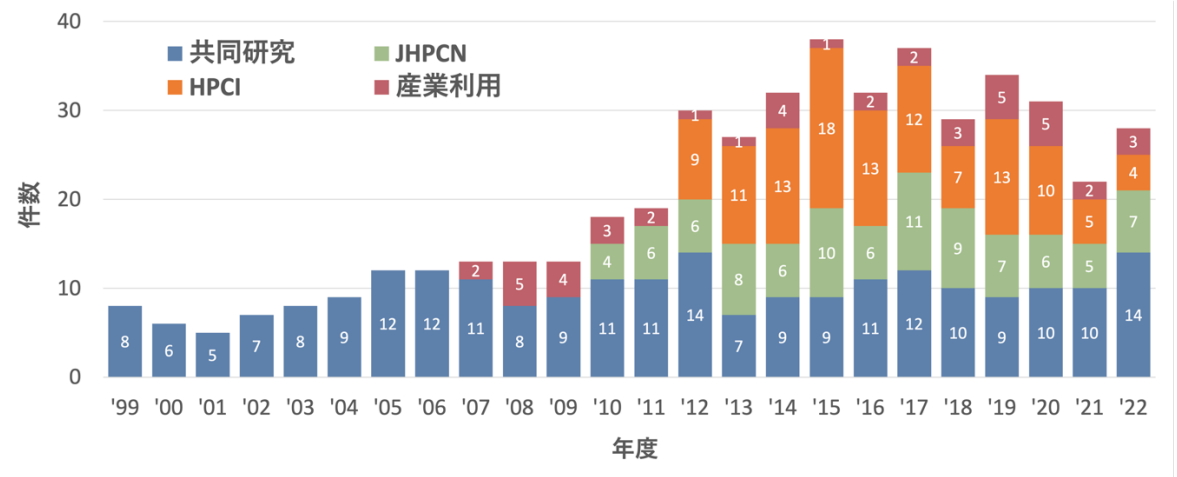

図 2.3-2　共同研究数の推移

また，近年では科学研究のみならず，社会基盤としてスーパーコンピュータの在り方に関する検討も精力的に進めている．東北大学災害科学国際研究所，NEC他と本センターの共同研究においては，津波浸水被害予測システムの開発に成功した．図2.3-3に示す本システムでは，地震情報の自動取得と津波発生・伝播・浸水・被害予測，結果の可視化・配信をリアルタイムで行うことで，いつ津波が発生しても迅速な浸水被害予測を可能にしている．また，本システムは内閣府総合防災情報システムの一機能として採用され，南海トラフ地震への備えとして，鹿児島県から静岡県までの6,000kmの海岸線を対象に2018年4月１日より実運用を開始しており，我が国の津波防災対策・対応の高度化と国土強靭化に貢献している．2022年度にはさらにシミュレーション領域を拡大し，太平洋側13,000kmおよび日本海側2,700kmの計15,700kmの海岸線を対象としている．

三井共同建設コンサルタント株式会社との共同研究においては，水災害・リスクマネジメント国際センター（ICHARM）が開発した降雨流出氾濫モデル（Rainfall-Runoff-Inundation，RRI）モデルの高速化および並列化に取り組んだ．RRIモデルは，従来のモデルでは独立に行われていた降雨による河川への流出現象の解析と，流出による氾濫現象の解析を一体的に行い，精度の高い氾濫予測を可能にする．一方，RRIモデルを広域に適用する場合，計算時間が大きな課題となっていた．本共同研究では，SX-Aurora TSUBASAを対象としてRRIモデルの高速化・並列化を実施し，大幅な高速化を実現した．本成果は，リアルタイムの予測降雨データを元に河川の水位や氾濫量を予測する氾濫予測システムの開発に活かされ，兵庫県をはじめとする導入自治体における水害対策に貢献している．

また，名古屋工業大学，日本気象協会と本センターの共同研究により，気象予報と人体の個体差を考慮した熱中症リスク評価システムを開発した．近年，今後の超高齢化社会の到来と加速する地球温暖化と相まって更なる搬送者数の上昇が予想されるなど，「熱中症」への取り組みが社会的関心事となっている．開発した熱中症リスク評価システムは，組織数51，解像度1mmの人体モデルを用いて人体の体温変化を気象（温度・湿度・日光照射量等），性別・年齢・体重・身長の違いによる体温上昇，発汗の相違など工学・物理学的な見地から熱中症発症リスクを評価可能にしている．これら成果は，図2.3-4に示す様に，日本気象協会推進「熱中症ゼロへ」の公式サイトにて個人ごとの熱中症の危険度を簡易的に診断する『熱中症セルフチェック』（https://www.netsuzero.jp/selfcheck）を通して，熱中症低減に向けた啓発活動に活用されている．

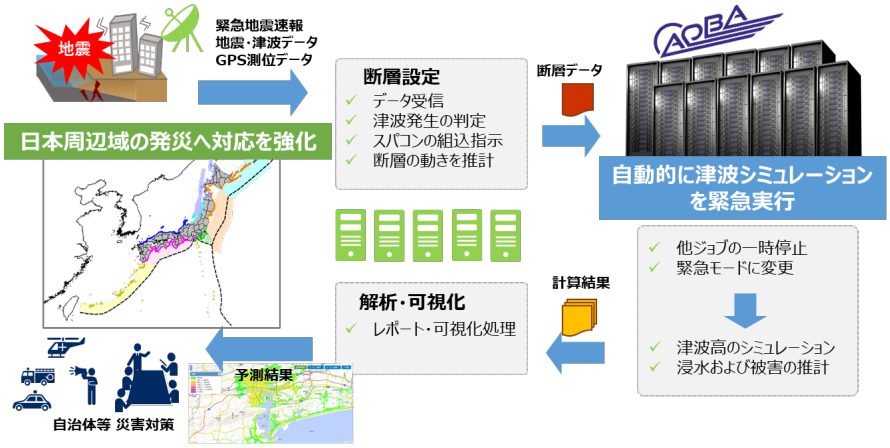

図 2.3-3　津波浸水被害予測システム

図 2.3-4　熱中症セルフチェック

2.4.　大規模科学計算システムの研究開発

本センターでは，高速化支援活動を通して得られたアプリケーションに関する臨床学的知見を本センターで運用している大規模科学計算システムの設計にフィードバックさせるべく，高性能計算システム設計に関する研究をマイクロアーキテクチャ，システムソフトウェアレベルの研究開発に精力的に取り組んでいる．次期大規模科学計算システム開発の要素技術としては，次世代ベクトルプロセッサ・システムの開発，高性能低消費電力を実現するメモリサブシステムに関する研究開発，再構成可能ハードウェアを活用した科学技術計算，性能可搬性を支援するプログラミングモデル，高効率運用のためのジョブスケジューリング機構に関する研究を推進した．

これらの研究成果は，学術論文誌や，SC，ISC，COOL Chips 等のスーパーコンピュータや，コンピュータ設計に関する国際会議の論文として毎年発表しており，国内外から高く評価されている．また，スーパーコンピュータのシステムの運用に関しても，外気導入や室温管理機構の改善や，SX-Aurora TSUBASA が具備する低消費電力モードを適材適所で活用する技術の開発にも取り組み，高い稼働率，システムスループットを維持しながら，大幅な運用コストの削減を実現している．

あわせて2006年より，ドイツシュトゥットガルト計算センター（HLRS）と共同で毎年2回高性能計算に関する国際ワークショップ（Workshop on Sustained Simulation Performance）の開催，SCや関連する国際会議におけるブース展示(図2.4-1)において，本センターの研究活動の成果を国内外に発信している．

図 2.4-1　SC22 におけるブース展示

これらの国内外で高く評価されている成果はいずれも，利用者・本センターの教職員・NEC の技術者が密に連携した高速化支援体制・共同研究体制が礎になっている．特に，高速化支援を遂行するためには，研究目的はもちろんその内容，利用者プログラムの計算アルゴリズムとデータ構造も熟知する必要がある．このために，利用者との打ち合わせを重ね，本研究に携わる者がこれらを理解し，大規模科学計算システムに適したアルゴリズム，プログラミング，データ構造について提案し，ユーザである計算科学者との共同研究を推進してきた．今後も将来の計算シミュレーションによるサイエンスの進歩，イノベーションの創出を加速するためにも，高速化推進研究活動に真摯に取り組んでいく所存である．

2.5.　まとめ

本章では，1997年より取り組んでいる高速化推進活動の取り組みと成果について述べた．これらの2018年12月から現在に至るまでの高速化推進研究活動報告については，本報告書第3章以降に詳細に説明する．最後に本高速化推進研究活動は，利用者の協力なしには為し得ない．ここにあらためて感謝の意を表する．

3. 高速化推進研究活動の成果

情報部デジタルサービス支援課　大泉健治　小野敏　山下毅　齋藤敦子　森谷友映

本センターのスーパーコンピュータシステムは，SX-ACE の運用を 2015 年 2 月に開始し，その後 2020 年 7 月末までサービスを行った．次いで 2020 年 10 月より AOBA-A（SX-Aurora TSUBASA）の運用を開始し，2022 年 10 月には，計算機資源の増強としてクラウドサービス AOBA-C（AOBA-A と同機種）の提供を開始し，2023 年 7 月までサービスを行った．

SX-ACE の本格的な運用の開始となる 2016 年度から 2017 年度にかけては，ジョブの実行ノード時間が 100,000 を超える大規模・長時間ジョブの割合が増加する傾向にあったが，2018 年度以降ノード時間が短いジョブ，すなわち小，中規模・短時間ジョブの割合が増加した．この傾向は AOBA-A の運用を開始した 2020 年度から 2022 年度までについても同様であり，従来の SX シリーズでは運用期間の後半になるほど実行ノード時間が大きくなる傾向，とは異なる状況であった．ジョブのノード時間分布について詳しくは 3.2 節および 3.3 節に記述した．

利用者ジョブの実行ノード時間の傾向変化に伴い，24 時間以内の演算時間が指定されたジョブのアサインを優先的に行うなど，より効率的なシステムの利用のための運用方針の検討と実施を行っているが，ベクトル演算器を主とする SX システムを高効率に利用するために，ベクトル演算率と並列化率を可能な限り高めることの重要性に変わりは無い．そのためにはコンパイラがコードの解析を基に自動で行うベクトル化・最適化および並列化の機能を活用すると共に，更なる性能向上のためには利用者がプログラムの高速化に積極的に関わる必要がある．そのため本センターでは 1997 年より現在まで継続して，利用者，センター教職員および計算機ベンダーが三者一体となった高速化支援活動を実施している．

以下では，2018 年度から 2022 年度までの高速化支援を行ったプログラムのうち，主なものについて概略を述べる．

3.1. 高速化推進研究活動（2018 年度～2022 年度）

高速化支援を行ったプログラムの主なものについて，表 3.1-1 に 2018 年度（平成 30 年度），表 3.1-2 に 2019 年度（令和元年度），表 3.1-3 に 2020 年度（令和 2 年度），表 3.1-4 に 2021 年度（令和 3 年度），表 3.1-5 に 2022 年度（令和 4 年度）の高速化支援による性能向上比と主な改善点を示す．

表中の性能向上比における演算性能は，ベクトル化促進，演算・メモリ性能最適化等による高速化前後の演算時間比を示す．並列性能は，MPI 並列化，自動並列化の促進，演算の隠蔽，通信の最適化等による高速化前後の演算時間比を示す．なお，並列化が未実施のプログラムの高速化を行った後，並列化による高速化を実施したプログラムは，性能向上比の両欄に記載がある．

表 3.1-1 2018 年度(平成 30 年度)の高速化支援性能向上比

プログラム番号	主な改善点	性能向上比	
		演算性能	並列性能
1	・作業配列の導入によるリストベクトルの解消 ・指示行によるメモリアクセス性能の改善 ・指示行による平均ベクトル長の改善	1.6 倍	
2	・作業配列の導入によるベクトル化の促進 ・演算順序の変更によるベクトル化の促進	1.8 倍	5.1 倍 (128 コア並列)
3	・MPI 分割の改善 ・不要な転送の削減 ・配列のアクセス連続化		2.1 倍 (32 コア並列)
4	・ベクトル実行不可の演算部分を VH Call で処理	1.8 倍	
5	・ループの入れ替え ・7/3 乗計算の CBRT 関数への置き換え	1.2 倍	
6	・ハイパープレーン法の 2 次元から 3 次元への変更によるループ長の拡大 ・コンパイラオプションにより除算を近似計算から除算命令に変更		1.1 倍 (636 コア並列)

表 3.1-2 2019 年度(令和元年度)の高速化支援性能向上比

プログラム番号	主な改善点	性能向上比	
		演算性能	並列性能
1	・指示行によるループ展開およびベクトル化の促進 ・明示的なインライン展開によるベクトル化の促進	24.9 倍	
2	・動的な配列領域の確保による省メモリ化とグローバルメモリの利用 ・MPI 分割数の調整による演算時間インバランスの解消 ・MPI 通信命令の最適化		1.4 倍 (128 コア並列)
3	・作業配列の導入によるベクトル化の促進 ・MPI 通信命令の最適化		1.8 倍 (576 コア並列)
4	・ASL ライブラリの変更による演算の効率化		4.9 倍 (4 コア並列)

5	・指示行によるループ展開およびベクトル化の促進 ・MPI 通信命令の最適化		6.7 倍 (32 コア並列)
6	・メモリアクセスを間接アクセスから連続アクセスに変更	2.8 倍	
7	・ループ分割, ループ入れ替え, 一次元配列の次元の拡張によるループ長の拡大 ・複数 DO ループのブロック化によるループ長の拡大 ・DO ループ内のスキップ処理に対応したマスク処理の適用		21.8 倍 (8 コア並列)
8	・メモリアクセスを間接アクセスから連続アクセスに変更		1.1 倍 (636 コア並列)
9	・FFTW3 ルーチンの変更(一次元 FFT→一次元多重 FFT) ・ベクトル演算率向上のためにループを分割 ・ベクトル実行不可の演算部分を VH Call で処理		2.5 倍 (16 コア並列)
10	・指示行によるベクトル化の促進 ・ループ入れ換えおよび最内ループ変数の定数化によるベクトル長の伸長 ・IF 文の簡略化による演算の効率化	7.9 倍	

表 3.1-3 2020 年度(令和 2 年度)の高速化支援性能向上比

プログラム番号	主な改善点	性能向上比	
		演算性能	並列性能
1	・指示行によるループ展開およびベクトル化の促進	3.5 倍	
2	・指示行によるループ展開およびメモリアクセスの効率化 ・明示的なインライン展開によるベクトル化の促進 ・コンパイルオプションによるループ融合の促進	10.9 倍	
3	・高速フーリエ変換部分の高速化(多重 FFT ルーチンへの置換)	2.3 倍	
4	・MPI 分散並列化による使用メモリの削減 ・MPI 並列化による大規模実行化		64VE で実行可能に
5	・ループインデックス計算の簡略化によるベクトル化の促進	4.3 倍	
6	・ASL ライブラリ FFT ルーチンの NLC FFT3 インタフェースルーチンへの置換	1.7 倍	

7	・ブロック化およびベクトルレジスタの使用によるメモリアクセス量の削減とベクトル化の促進	1.7 倍	

表 3.1-4 2021 年度(令和 3 年度)の高速化支援性能向上比

プログラム番号	主な改善点	性能向上比	
		演算性能	並列性能
1	・指示行によるメモリアクセスの効率化 ・コンパイルオプションによる自動並列化の適用 ・明示的インライン展開によるサブルーチン呼び出しコストの削減		1.8 倍
2	・3 重ループの演算順序固定によるメモリアクセスの効率化 ・インライン展開によるサブルーチン呼び出しコストの削減	2.9 倍	
3	・バイナリデータファイル I/O の最適化	3.1 倍	
4	・ベクトル実行不可の演算部分を VH Call で処理	2.5 倍	
5	・MPI 2 次元分割への変更による大規模データ、多ノード数実行における実効性能の向上 ・FFT 計算他での性能最適化(通信と演算のオーバーラップ等)		1.6 倍
6	・numpy ライブラリへの変更 ・冗長な処理に対する効率的な枝刈りによる演算コストの削減	6.7 倍	

表 3.1-5 2022 年度(令和 4 年度)の高速化支援性能向上比

プログラム番号	主な改善点	性能向上比	
		演算性能	並列性能
1	・ベクトル実行不可の演算部分を VH Call で処理		2.5 倍
2	・MPI2 次元分割大規模演算コードの MPI タスク配置の制御および最適化を実施		1.5 倍
3	・OpenMP の手動スレッド操作によるオーバーラップ手法チューング ・プログラムコードの構造精査によるコード効率化		1.5 倍

4	・ループ分割およびループ入れ替えによるベクトル化の促進		140 倍
5	・自動並列と MPI のハイブリッド並列化チューニングによる 通信コストの削減		1.4 倍

3.2. スーパーコンピュータ SX-ACE のベクトル化・並列化の状況

2018 年度から 2020 年度 7 月までの各年度における，SX-ACE システムで実行されたジョブのベクトル演算率および並列化率と，ジョブのノード時間が全ノード時間に占める割合（ノード時間割合）との関係を図 3.2-1 に示す．なお，ベクトル演算率はジャーナル情報から，並列化率は CPU 時間合計を実行ノード数と 1 ノード内のコア数 4 で除した値とし，ノード時間はジョブの経過時間と実行ノード数の積とした．

2020 年度においてはベクトル演算率と並列化率がともに 90% 以上となるジョブのノード時間割合が 60% に達しており，利用者ジョブの SX-ACE システムへの最適化，特に並列性能の向上が見られた．

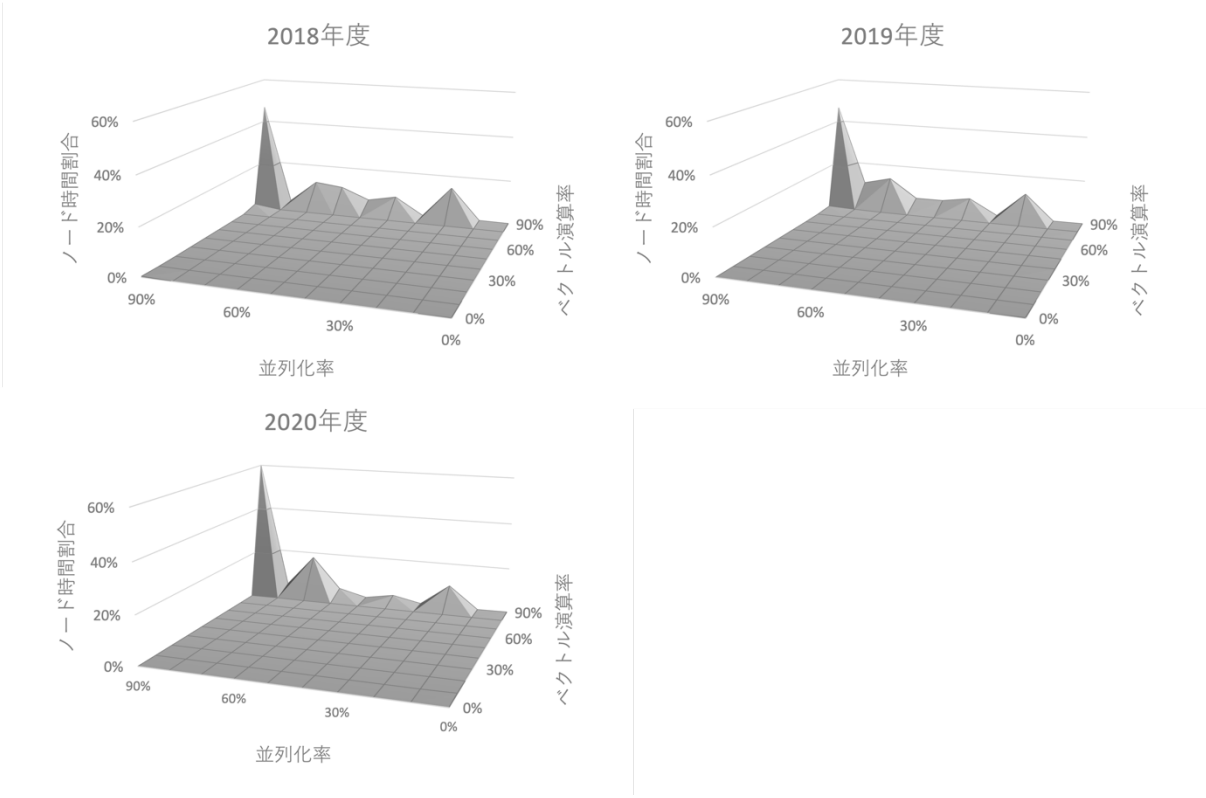

図 3.2-1 ベクトル演算率および並列化率とノード時間割合

3 年間の利用者ジョブのベクトル演算率と並列化率の状況を表 3.2-1 に示す．表 3.2-1 は SX-ACE で実行された利用者ジョブをベクトル演算率と並列化率で分類し，そのノード時間の割合を全ノード時間に対して百分率で表したものである．また，表 3.2-1 からベクトル演算率と並列化率に関して 3 つのカテゴリで分類したものを表 3.2-2 に示す．

表 3.2-1 ベクトル演算率と並列化率の状況 [%]

ベクトル演算率	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
90%	0.2	0.1	12.7	0.1	6.9	4.6	6.9	13.2	5.8	48.0
80%	0	0	1	0	0	0	0	0	0.1	1.0
70%	0	0	0	0	0	0	0	0	0	0
60%	0	0	0	0	0	0	0	0	0	0
50%	0	0	0	0	0	0	0	0	0	0
40%	0	0	0	0	0	0	0	0	0	0
30%	0	0	0	0	0	0	0	0	0	0
20%	0	0	0	0	0	0	0	0	0	0
10%	0	0	0	0	0	0	0	0	0	0.1
0%	0.1	0	0	0	0	0	0	0	0	0.2

並列化率

表 3.2-2 ベクトル演算率と並列化率の状況

ベクトル演算率と並列化率の分類	ノード時間の割合
ベクトル演算率と並列化率の双方が 90% 以上のジョブ	48.0%
ベクトル演算率が 90% 以上のジョブ	98.5%
並列化率が 90% 以上のジョブ	49.3%

表 3.2-2 に示したように，ベクトル演算率が 90% を超えるジョブのノード時間が 98.5% を占めており，SX-ACE の運用開始当初に見られたベクトル演算率がほぼ 0% のノード時間割合が 15% あった状況と比較すると，高速化支援等によりジョブのシステムへの最適化が進んだことによる効果が確認できる．

表 3.2-3 は SX-ACE の利用が本格化した 2016 年度から 2020 年度までの 5 年間について，ジョブのノード時間ごとのジョブ数の割合を示したものである．2016 年度および 2017 年度においては 100,000 ノード時間以上のジョブの割合が 30% を超える状況であったが 2018 年度以降は減少に転じ，ジョブの小・中規模化への顕著な推移が見られた．

表 3.2-3 ノード時間使用分布

ノード時間	2016 年度	2017 年度	2018 年度	2019 年度	2020 年度
0 ～ 999	17%	20%	19%	27%	37%
1,000 ～ 9,999	28%	19%	21%	31%	34%
10,000 ～ 49,999	11%	17%	29%	23%	18%
50,000 ～ 99,999	11%	10%	13%	12%	8%
100,000 ～	33%	34%	18%	7%	3%

3.3. サブシステム AOBA-A・AOBA-C（SX-Aurora TSUBASA）のベクトル化・並列化の状況

2020 年度から 2022 年度までの各年度における，サブシステム AOBA-A で実行されたジョブのベクトル演算率および並列化率と，ジョブのノード時間が全ノード時間に占める割合（ノード時間割合）との関係を図 3.3-1 に示す．なお，ベクトル演算率はジャーナル情報から，並列化率は VE の CPU 時間合計を

実行 VE 数と 1VE 内のコア数 8 で除した値とし，ノード時間はジョブの経過時間と実行 VE 数の積とした．また，2022 年 10 月に運用を開始したクラウドサービス AOBA-C の状況は 2022 年度のグラフに含む．

AOBA-A を導入した 2020 年度において，既にベクトル演算率と並列化率がともに 90% 以上となるジョブのノード時間割合が 60% に達しており，実行されたジョブは前システムの SX-ACE からの移行がスムーズに行われたことを示している．続く 2021 年度ではベクトル演算率は 90% を超えるが，並列化率が低いジョブのノード時間割合が増加している．これらのジョブは，研究室の PC やサーバで実行されていたプログラムで，並列化が未実施のプログラムのジョブと推察される．このような新規のユーザコードで，小・中規模なジョブのノード時間が増加しているのが近年の特徴で，これらコードに対する高速化支援も重要な点となっている．2022 年度ではベクトル演算率と並列化率ともに 90% を超えるジョブのノード時間割合が増加していることは本活動の成果であると言える．

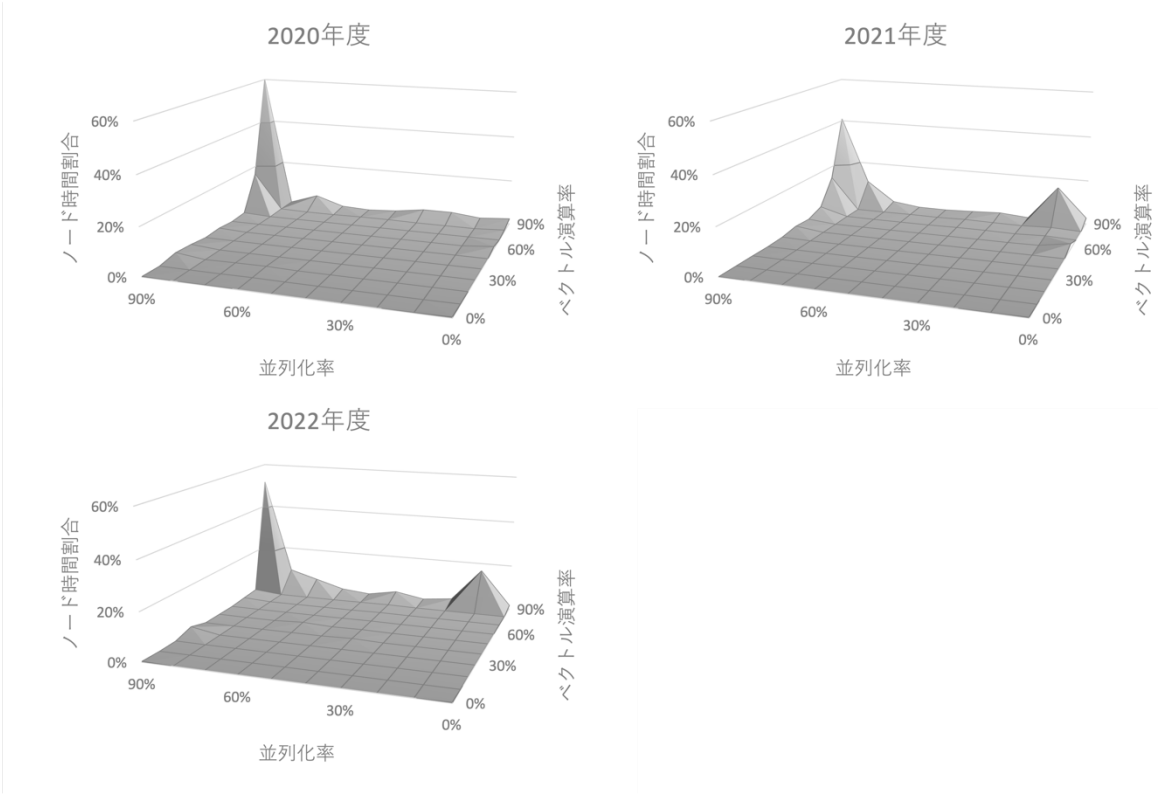

図 3.3-1 ベクトル演算率および並列化率とノード時間割合

3 年間の利用者ジョブのベクトル演算率と並列化率の状況を表 3.3-1 に示す．表 3.3-1 は AOBA-A で実行した利用者ジョブをベクトル演算率と並列化率で分類し，そのノード時間の割合を全ノード時間に対して百分率で表したものである．また，表 3.3-1 からベクトル演算率と並列化率に関して 3 つのカテゴリで分類したものを表 3.3-2 に示す．

表 3.3-1 ベクトル演算率と並列化率の状況［%］

ベクトル演算率	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
90%	2.0	15.9	1.4	0.9	2.0	0.5	1.3	4.1	9.6	48.0
80%	0	0.2	0	0.1	0.3	0	0	0	0	6.6
70%	0	0.2	0	0.1	0.2	0	0	0	0	1.7
60%	1.0	0.1	0	0	0	0	0	0	0	0.5
50%	0	0	0	0	0.1	0	0.4	0	0	0.6
40%	0	0	0	0	0.3	0	0	0	0	0.2
30%	0	0	0	0	0	0	0	0	0	1.3
20%	0	0	0	0	0	0	0	0	0	0.1
10%	0	0	0	0	0	0	0	0	0	0
0%	0.1	0	0	0	0	0	0	0	0	0.2

並列化率

表 3.3-2 ベクトル演算率と並列化率の状況

ベクトル演算率と並列化率の分類	ノード時間の割合
ベクトル演算率と並列化率の双方が 90% 以上のジョブ	48.0%
ベクトル演算率が 90% 以上のジョブ	85.7%
並列化率が 90% 以上のジョブ	59.2%

表 3.3-1 および表 3.3-2 の状況から，AOBA-A の運用開始から 2 年半，および AOBA-C の運用開始から 6 ヶ月の期間では，ベクトル演算率が 90% 以上のジョブのノード時間割合が 85.7% と低い一方，並列化率が 90% 以上のジョブの割合が 59.2% と，過去のシステムでの状況と比較して並列化率が高い状況であった．このことは，他システムで高度に並列化されたコードが実行されるようになったと推察される．このような利用者コードの，ベクトル演算率を高める高速化支援が急務である．

表 3.3-3 は 2020 年度から 2022 年度までの 3 年間について，ジョブのノード時間ごとのジョブ数の割合を示したものである．なお，SX-ACE システムの総ノード数が 2,560 ノードであるのに対して，AOBA-A の総 VE 数が 576VE であることを考慮して，ノード時間の範囲は表 3.2-3 の 1/5 に変更している．

2020 年度から 2021 年度にかけては，中規模なジョブから 20,000 ノード時間以上の大規模なジョブへの移行が見られる状況であったが，2022 年度では大規模なジョブの割合が減少に転じ，大規模なジョブと，小・中規模なジョブの 2 分化が見られるようになった．

表 3.3-3 ノード時間使用分布

ノード時間	2020 年度	2021 年度	2022 年度
0 ～ 199	12%	25%	21%
200 ～ 1,999	36%	32%	46%
2,000 ～ 9,999	4%	13%	18%
10,000 ～ 19,999	35%	1%	2%
20,000 ～	13%	29%	13%

3.4. 今後の取り組み

3.2節および3.3節で述べたように，SX-ACEの運用期間中はジョブの大規模化・長時間化が進んだ一方，AOBA-Aの2年半の運用期間では小・中規模なジョブと大規模なジョブの2分化が見られるようになった．計算機の効率的な運用には，ユーザジョブのさらなる高いベクトル演算率と並列化率での実行を実現するためのMPI並列化を含む高速化支援が今後も必要である．このことに加え，利用者ジョブの実行傾向に応じてバッチジョブのスケジューリングや，ジョブの実行要求時間に応じたノードの提供方式を検討することは，限られた計算機資源の上で利用者の待ち時間を減らすなど，利用者支援として不可欠である．

4. 大規模科学計算システムの構成

スーパーコンピューティング研究部　滝沢寛之　高橋慧智　下村陽一
情報部デジタルサービス支援課　大泉健治　小野敏　山下毅　齋藤敦子　森谷友映
高性能計算技術開発(NEC)共同研究部門　撫佐昭裕　磯部洋子　曽我隆　山口健太
日本電気株式会社　加藤季広
NEC ソリューションイノベータ株式会社　佐藤佳彦

本センターでは，2020 年 10 月よりスーパーコンピュータシステム AOBA の運用を開始した．AOBA はサブシステム AOBA-A(SX-Aurora TSUBASA B401-8)およびサブシステム AOBA-B(LX 406Rz-2)の 2 種類の計算機システムと，ストレージシステムで構成される．ストレージシステムは，高速かつ高密度ストレージである DDN SFA7990XE を導入している．ファイルシステムは，NEC 製の分散・並列ファイルシステムである Scalable Technology File System(ScaTeFS)で構成され，実効容量 2PB のユーザホーム領域を提供している．さらに，2022 年 10 月からは，計算機資源の増強のために期間限定でクラウドサービス AOBA-C(SX-Aurora TSUBASA B302-8)の提供を開始した． 図 4-1 に，2022 年 10 月時点における本センターのシステム構成を示す．

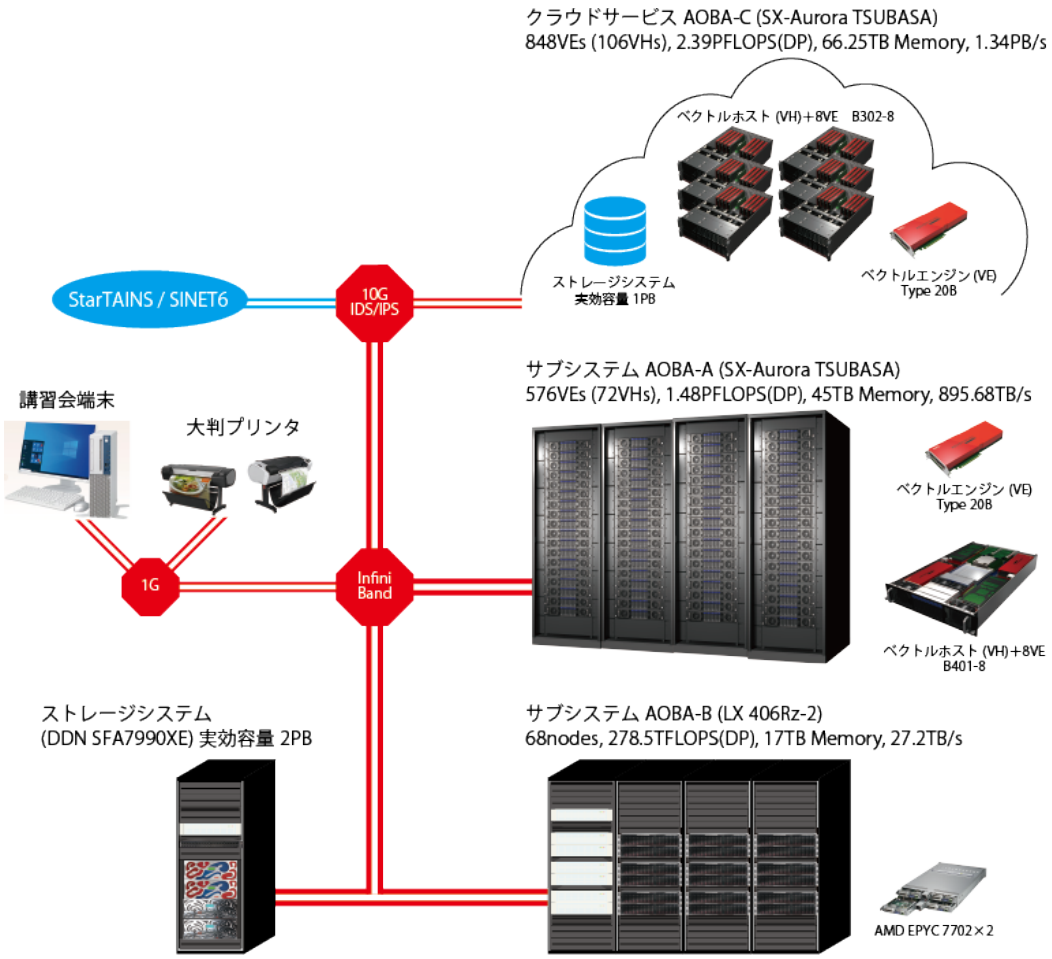

図 4-1　2022 年 10 月時点におけるシステム構成

さらに，2023 年 8 月からはサブシステム AOBA-S（SX-Aurora TSUBASA C401-8）の運用が開始された．AOBA-S は SX-Aurora TSUBASA の最新機種である VE30 を搭載しており，AOBA-A の 7 倍となる 504VH で構成されている．図 4-2 に 2023 年 8 月時点におけるシステム構成を示す．AOBA-S の運用開始に伴い，クラウドサービスである AOBA-C の運用は終了した．

AOBA-S は AOBA-A, AOBA-B とは独立したシステムとして運用されている．利用者は AOBA-S 用のフロントエンドサーバにログインし，AOBA-S にジョブを投入する．AOBA-S 用のストレージシステムとして DDN ES400NVX2 を新たに導入した．ファイルシステムは分散並列ファイルシステムである Lustre を採用しており，実効容量は 4.5 PB となる．

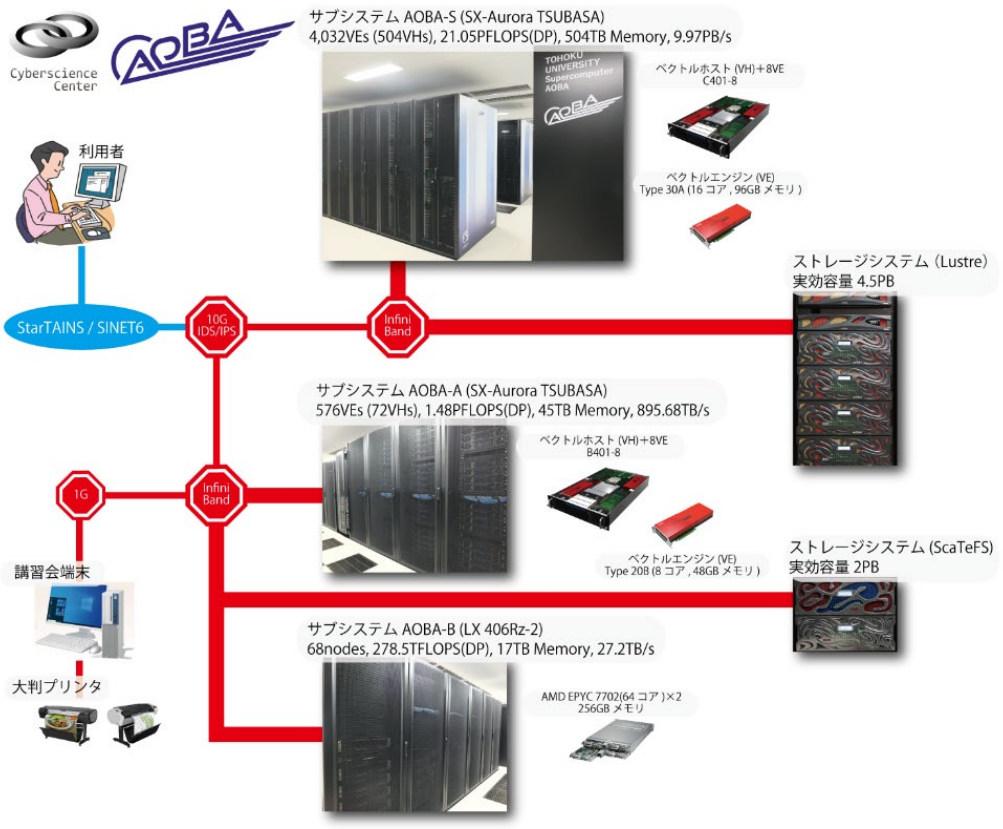

図 4-2 2023 年 8 月時点におけるシステム構成

4.1. AOBA-A, AOBA-C および AOBA-S の特徴

AOBA-A, AOBA-C および AOBA-S を構成する SX-Aurora TSUBASA は Vector Engine（以降 VE）と呼ばれる PCI Express カードにベクトルプロセッサと主記憶を搭載し，これを Vector Host（以降 VH）と呼ばれる標準的な x86 サーバに接続することにより構成されている．AOBA-A および AOBA-C は第二世代の VE（Type 20B）を搭載しており，AOBA-S は 2023 年 8 月時点で最新の第三世代の VE（Type 30A）を搭載している．システムの主な諸元を表 4.1-1 に示す．

表 4.1-1 AOBA-A, AOBA-C および AOBA-S の主要諸元

		AOBA-A	AOBA-C	AOBA-S
Vector Engine	モデル名	Type 20B	Type 20B	Type 30A
	ベクトルコア数	8	8	16
	理論ベクトル演算性能	2.45 TFlop/s	2.45 TFlop/s	4.91 TFlop/s
	メモリ容量	48 GB	48 GB	96 GB
	メモリ帯域	1.53 TB/s	1.53 TB/s	2.45 TB/s
	LLC 容量	16 MB	16 MB	64 MB
	LLC 帯域	3.0 TB/s	3.0 TB/s	6.4 TB/s
	L3 キャッシュ容量	-	-	2 MB
Vector Host	CPU モデル名	AMD EPYC 7402P	Intel Xeon Gold 6326	AMD EPYC 7763
	CPU コア数	24	16	64
	メモリ容量	256 GB	256 GB	256 GB
	搭載 Vector Engine 数	8	8	8
	ネットワーク I/F	InfiniBand HDR x2	InfiniBand HDR x2	InfiniBand NDR200 x2
システム	Vector Host 数	72	106	504
	Vector Engine 数	576	848	4,032
	総理論演算性能	1.48 PFlop/s	2.39 PFlop/s	21.05 PFlop/s
	総メモリ帯域	895.68 TB/s	1.34 PB/s	9.97 PB/s
	総メモリ容量	45 TB	66.25 TB	504 TB

4.1.1. VH-VE 構成

ここでは，現在運用中の AOBA-A と AOBA-S の VH-VE 構成について概説する．

図 4.1.1-1 に AOBA-A の VH-VE 構成（左）と VE（Type 20B）のブロック図（右）を記載する．AOBA-A の構成要素である SX-Aurora TSUBASA B401-8 は VH と，8 枚の VE で構成される．VH のプロセッサは AMD EPYC 7402P であり 1socket 24 コアを備えている．プロセッサは 4 つの PCI Express Switch（PCIe SW）と 2 つの InfiniBand HCA カード（IB HCA）と直接接続されている．PCIe SW はそれぞれ 2 つの VE と接続されており，VE は 8 個の CPU コアと，コア間で共有される LLC（Last Level Cache），6 個の HBM2 メモリで構成される．

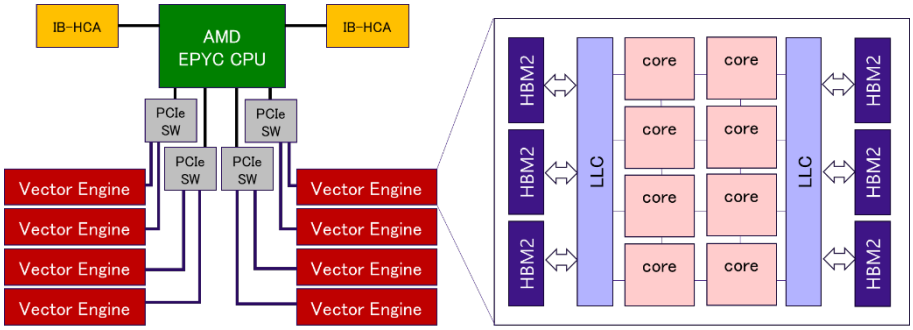

図 4.1.1-1 AOBA-A の VH-VE 構成の概要図

図 4.1.1-2 に AOBA-S の VH-VE 構成(左)と VE(Type 30A)のブロック図(右)を記載する. AOBA-S の VH のプロセッサは AMD EPYC 7763 であり 1socket 64 コアを備えている. AOBA-A と同様に, プロセッサは 4 つの PCIe SW を介して 8 つの VE と接続されている. VE は 16 個の CPU コア, LLC, 6 つの HBM2E メモリで構成され, AOBA-A の VE(Type 20B)と比較して, CPU 数は 2 倍, LLC の容量は 4 倍, メモリ容量は 2 倍の増加となる. 性能向上としては, VE あたりの理論演算性能は 2 倍, LLC の帯域幅は 2.13 倍, メモリ帯域幅は 1.6 倍となる. さらに, それぞれの CPU コアには新しく 2MB の L3 キャッシュ(L3C) が導入されている. L3 キャッシュにキャッシュするデータはソフトウェア制御(指示行の挿入)によりユーザ側で指定することができるため, キャッシュ容量を効率的に利用することができる. このようなメモリシステムの大幅な改善により, メモリ負荷の高いアプリケーションの性能が大幅に改善することが期待される. さらに, VE(Type 30A)では, パックドベクトル命令利用時のデータアラインメント制限の緩和, 間接参照によるベクトル総和演算を LLC 上の専用ハードウェアで実行する VLFA 命令の追加といった改善も加わっている.

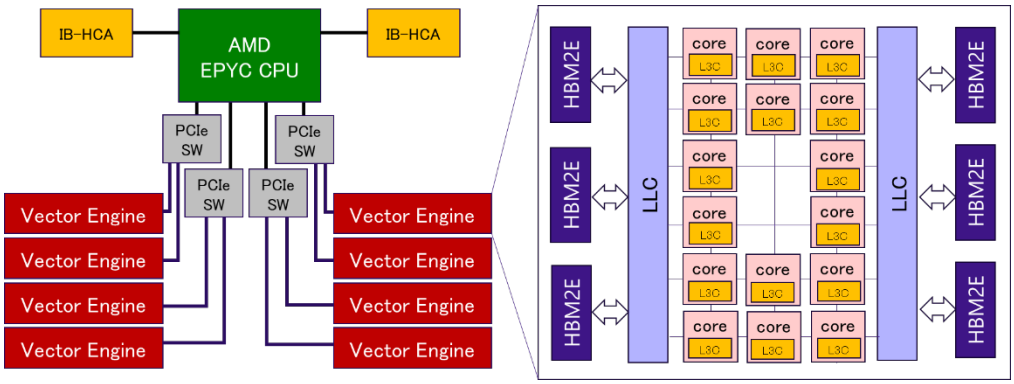

図 4.1.1-2 AOBA-S の VH-VE 構成の概要図

4.1.2. マルチノード構成

図 4.1.2-1 に AOBA-A のマルチノード構成図を記載する. AOBA-A は 72 台の VH で構成され, InfiniBand HDR による 2 段 Fat Tree ネットワークにより接続される. フルバイセクションバンド幅, ノンブロッキング構成によりすべての VH, ストレージ, 周辺サーバ群で高速な通信が可能となる.

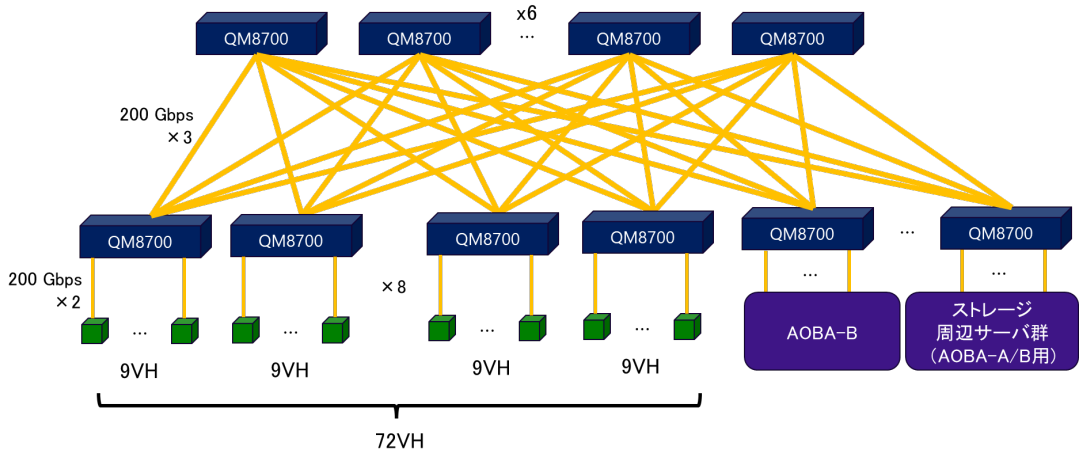

図 4.1.2-1 AOBA-A のマルチノード構成図

図 4.1.2-2 に AOBA-S のマルチノード構成図を記載する. AOBA-S は 504 台の VH で構成される. AOBA-A と同様にノード間接続ネットワークは 2 段 Fat Tree 構成となっている. それぞれの VH は 2 つの IB HCA を搭載しており, InfiniBand NDR200 によりノード間ネットワークに接続される. ネットワークスイッチ間は InfiniBand NDR により接続され, フルバイセクションバンド幅, ノンブロッキング構成で VH と接続される. AOBA-S 用のストレージや周辺サーバ群もこのネットワークに接続されており, VH とストレージ間での高速なデータ転送が可能となる.

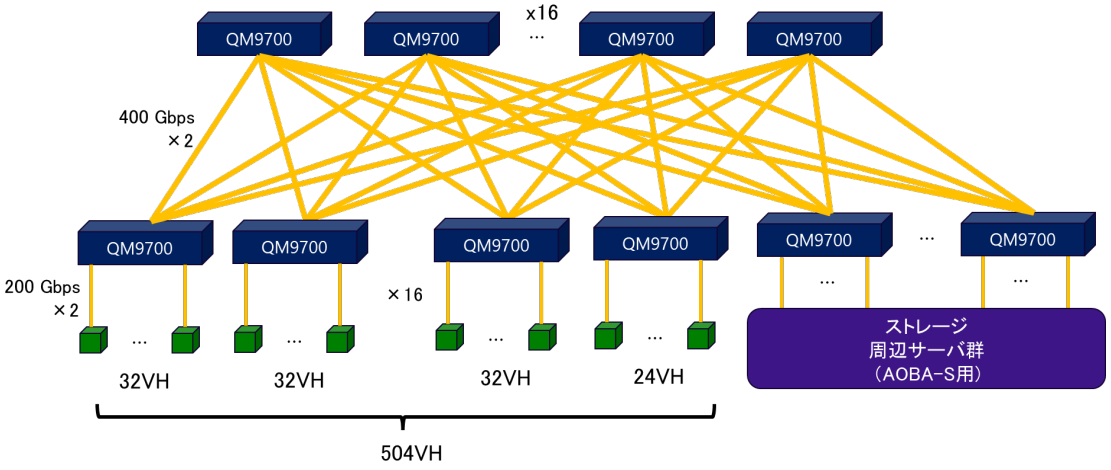

図 4.1.2-2 AOBA-S のマルチノード構成図

4.2. AOBA-B の特徴

AOBA-B は LX406Rz-2 で構成される. LX406Rz-2 は, 1 ノードに AMD EPYC 7702 (64 コア) を 2 基と, 256GB の主記憶を搭載している. 本センターでは, 68 ノードの LX406Rz-2 を運用しており, 総理論演算性能 278.5TFlop/s, 総メモリ容量は 17TB を有している. システムの主な諸元を表 4.2-1 に示す.

表 4.2-1 LX406Rz-2 の主要諸元

ノード	CPU 数	2
	コア数	128
	理論演算性能	4.096 TFlops/s
	メモリ容量	256 GB
	メモリ帯域	409.6 GB/s
システム	ノード数	68
	CPU 数	136
	コア数	8,704
	理論演算性能	278.5 TFlop/s
	メモリ容量	17 TB
	メモリ帯域	27.2 TB/s

4.2.1. ノード構成

図 4.2.1-1 に, LX407Rz-2 のノード構成を示す. LX406Rz-2 は, AMD EPYC 7702 プロセッサを 2 基搭載している. プロセッサあたり 64 コアが搭載されており, プロセッサ間は 4 チャネルの xGMI (inter-chip Global Memory Interconnect)インタフェースで接続されている. プロセッサあたり 8 チャネルの DDR4-3200 対応メモリインタフェースを装備しており, 16GB DIMM を使用することでノードあたり 256GB のメモリ容量を実装している. メモリバンド幅は 1 チャネルあたり 25.6GB/s であり, ノード全体では 409.6GB/s となる.

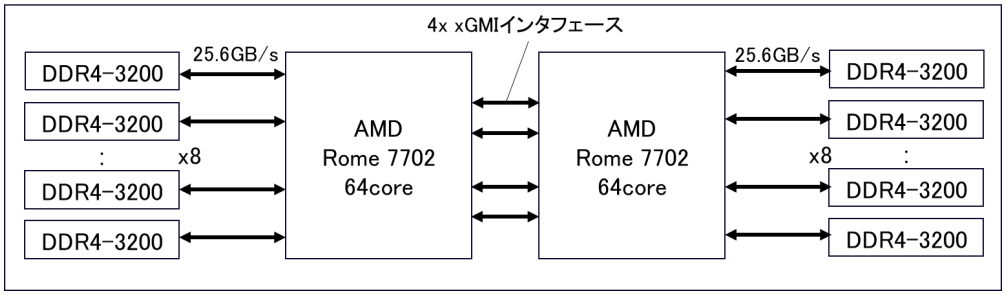

図 4.2.1-1 LX406Rz-2 のノード構成図

4.2.2. マルチノード構成

図 4.2.2-1 に LX406Rz-2 のマルチノード構成図を示す. AOBA-A と同様に InfiniBand HDR による 2 段 Fat-Tree ネットワークによりノード間が接続されており, フルバイセクションバンド幅, ノンブロッキング構成によりすべての演算サーバ, ストレージ, 周辺サーバ群で高速な通信が可能となる. 全ノードを使用することで最大 8,704 コアを使用したマルチノードプログラムの実行が可能である. ただし, センターの通常運用においては, 最大利用可能ノード数は 16 ノードまでとしているため, 1 ジョブあたり最大 2,048 コアが利用可能である.

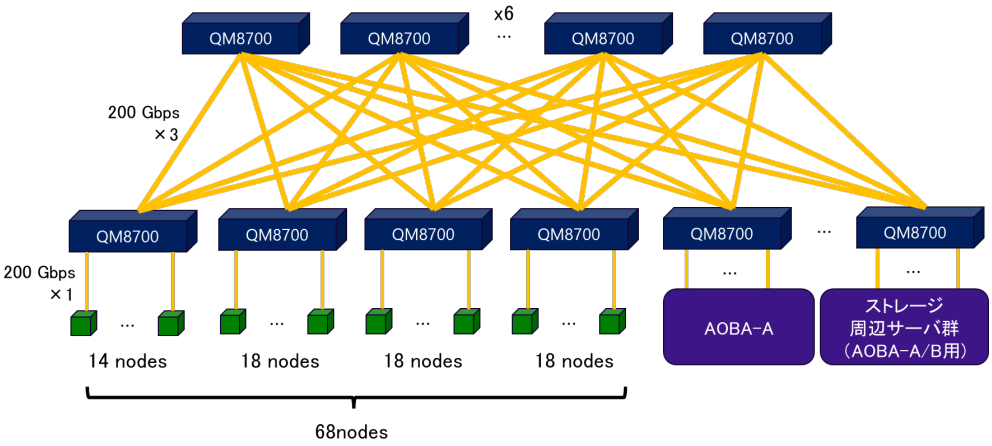

図 4.2.2-1 LX406Rz-2 のマルチノード構成図

5. スーパーコンピュータ AOBA-A の高速化の概要

スーパーコンピューティング研究部　滝沢寛之　高橋慧智　下村陽一
情報部デジタルサービス支援課　大泉健治　小野敏　山下毅　齋藤敦子　森谷友映
高性能計算技術開発(NEC)共同研究部門　撫佐昭裕　磯部洋子　曽我隆　山口健太
日本電気株式会社　加藤季広

5.1. ベクトル処理による高速化

5.1.1. ベクトル処理の概要

変数や配列の各要素のことをスカラデータとよぶ. これに対して, 行列の行要素や列要素など規則的に並んだデータ列のことをベクトルデータとよぶ. スカラ処理は, スカラデータを 1 つずつ処理する(スカラ命令)のに対して, ベクトル処理では 1 つの実行命令で, 規則的に並んだ複数の配列データ(ベクトルデータ)に対して同時に演算を行う(ベクトル命令)ことができる. 図 5.1.1-1 に, スカラ命令とベクトル命令の動作のイメージ図を記載する. 図の左側に DO ループのコードイメージを記載しているが, この DO ループは A(I)=B(I)+C(I), D(I)=E(I)+F(I)の 2 つの計算式を 200 回繰り返す. この DO ループをスカラ命令で処理する場合, 図の右側の上のように, まずは, I=1 の場合について, B(1)+C(1)の計算を行い, 次に E(1)+E(1)の計算を行う. I=1 の計算が完了すると次に I=2 の場合の処理として B(2)+C(2), E(2)+F(2)の計算を行うというように, DO ループの繰り返し変数 I が 200 になるまで逐次的に実行する. 一方で, 同じ DO ループをベクトル命令で実行する場合, ループ変数 I が 1～200 について 1 回のベクトル命令で計算することができるため, スカラ命令に比べて高速に計算することができる. なお, AOBA-A に搭載されたベクトルプロセッサの場合, 一度のベクトル命令で処理可能な要素数は最大 256 要素である. そのため, ループ長 200 のループの場合, B(I)+C(I) および E(I)+F(I)の計算はそれぞれ 1 回のベクトル命令で実行される. AOBA-A の性能を十分に引き出すためには, なるべく多くの処理をベクトル命令で実行する必要がある.

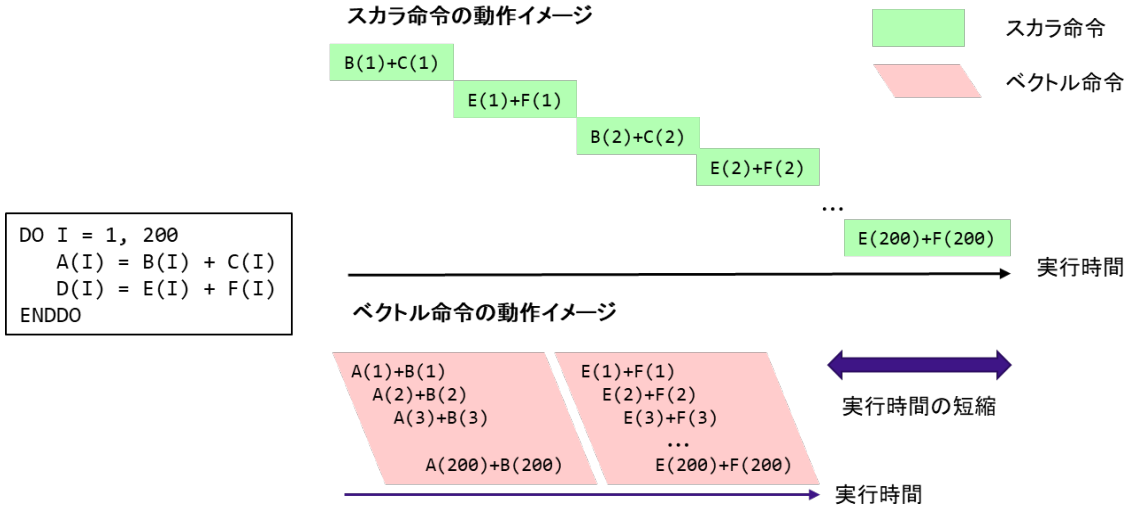

図 5.1.1-1 スカラ命令とベクトル命令の動作のイメージ図

5.1.2. ベクトル化率

プログラムを実行する場合，一般的にはプログラム中の全ての処理をベクトル処理として実行することは難しい．例えば，計算に必要なデータを読み込んだり，計算結果を書き出したりする入出力処理を含むループや，ループ内の計算が計算順番に依存している（計算順番が変わると計算結果が変わる）ようなループは，ベクトル命令が適用できないためスカラ処理となる．ベクトル化率（vector processing rate）とは，プログラムがどのくらいベクトル処理されているかを示す指標であり，あるプログラムをすべてスカラ処理で実行したときの総実行時間に対して，ベクトル処理が可能な部分の実行時間の割合をあらわす．図 5.1.2-1 にプログラムの実行時間の概念図を示す．

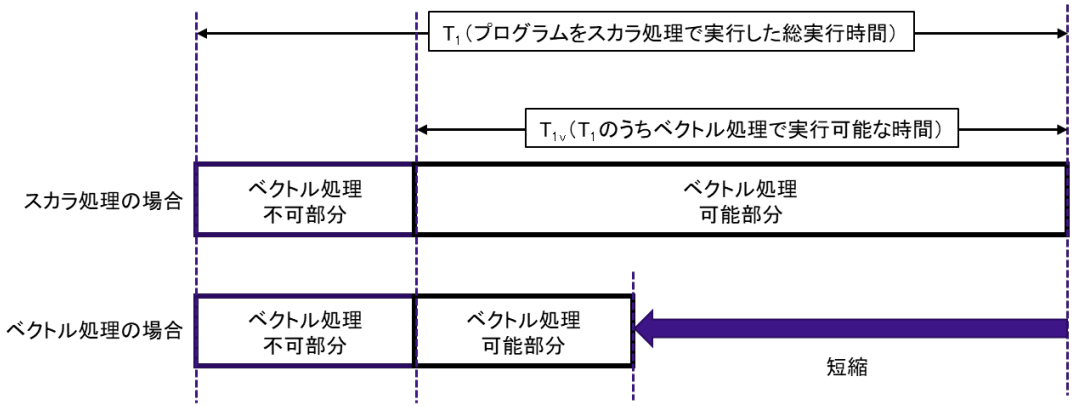

図 5.1.2-1 プログラムの実行時間の概念図

プログラム全体をスカラ処理で実行した時間を T_1，そのうち，ベクトル処理で実行可能な時間を T_{1v} とすると，ベクトル化率 α は次式で定義される．

$$\alpha = \frac{T_{1V}}{T_1} \times 100$$

ベクトル処理による性能向上比 P は，スカラ処理性能とベクトル処理性能の比を β とすると以下の式であらわすことができる．

$$P = \frac{100}{(100-\alpha)+\frac{\alpha}{\beta}}$$

この関係式をアムダールの法則とよび，図 5.1.2-2 はベクトル化率と性能向上の関係をあらわしている．この図から，高い性能向上を実現するためには，ベクトル化率をできるだけ 100% に近づけることが非常に重要であることがわかる．

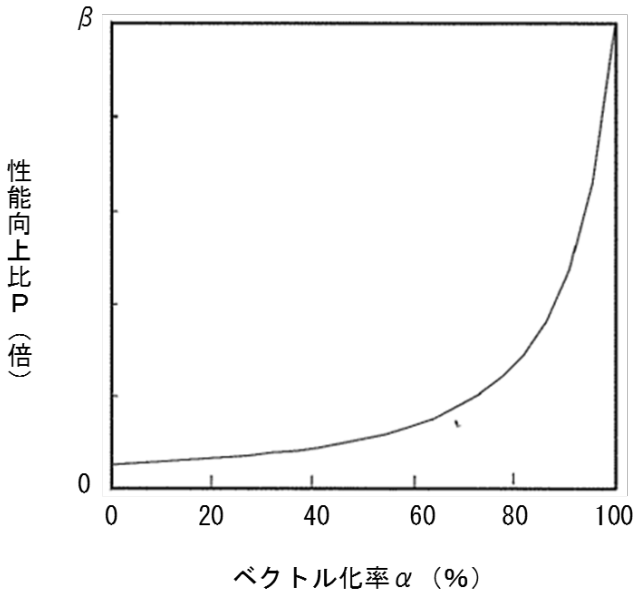

図 5.1.2-2 ベクトル化率と性能向上比の関係(アムダールの法則)

ベクトル化率を調べるためにはプログラムをすべてスカラ処理で実行した場合と，ベクトル処理可能な部分をベクトル化して実行した場合の実行時間が必要であるが，これらの時間は同時に得ることはできないためベクトル化率を容易に算出することは難しい．そこで，AOBA-A ではベクトル化率の代わりにベクトル演算率(vector operation ratio)という指標を使用する．ベクトル演算率は，プログラムで処理される全演算要素数に対するベクトル処理される演算要素数の割合で算出され，ほぼベクトル化率とみなせる指標である．なお，AOBA-A で利用可能な性能解析ツールを使用することで，利用者は容易に自分のプログラムのベクトル演算率を確認することができる．

5.1.3. ベクトル長

ベクトル化対象のループの繰り返し回数のことをベクトル長(ループ長)と呼ぶ．AOBA-A では 1 回のベクトル命令で最大 256 要素のベクトル処理を行うことができる．例えば，図 5.1.3-1 のような計算があった場合，a(i) + b(i) の計算は，2 回のベクトル命令で実行可能である．1 回目のベクトル命令で 256 要素同士の足し算を行い，2 回目のベクトル命令で残りの 44 要素同士の足し算を行う．1 回のベクトル命令で行った演算要素数の平均値を平均ベクトル長(average vector length)と呼び，この値はベクトル処理が効率的に行われているかを判断するのに重要な指標である．この例における平均ベクトル長は 150(=(256+44)/2)となる．ループ長が長くなればなるほど，平均ベクトル長は最大値の 256 に近づくため，できるだけベクトル化対象ループのループ長を長くすることが重要である．

```
do i=1,300
  a(i)=a(i)+b(i)
enddo
```

図 5.1.3-1 ループ例

ループ処理をベクトル化することで，必ず性能が向上するとは限らない．例えば，ループ長が非常に短いループをベクトル化した場合，逆に性能が悪化する場合もある．これには，ベクトル処理の立ち

上がり時間が関係している. ループをベクトル処理する場合, ベクトル処理が開始されるまで少し時間がかかり, これを立ち上がり時間とよぶ. 図 5.1.3-2 はループ処理をベクトル化した場合としない場合のループ長と実行時間の関係を示している. ベクトル処理した場合, ループ長が短いところでは, 立ち上がり時間の影響でベクトル化しない場合の性能が高くなっていることがわかる. ベクトル化した場合としない場合とで実行時間が同じになるループ長のことを交差ループ長と呼び, ループ長が交差ループ長(3 程度)より短い場合には, ベクトル化することで性能が悪化してしまう場合がある. 逆に, ループ長が長ければ長いほど, ベクトル化した場合の効果は大きくなるため, この図からもループ長を長くすることの重要性がわかる.

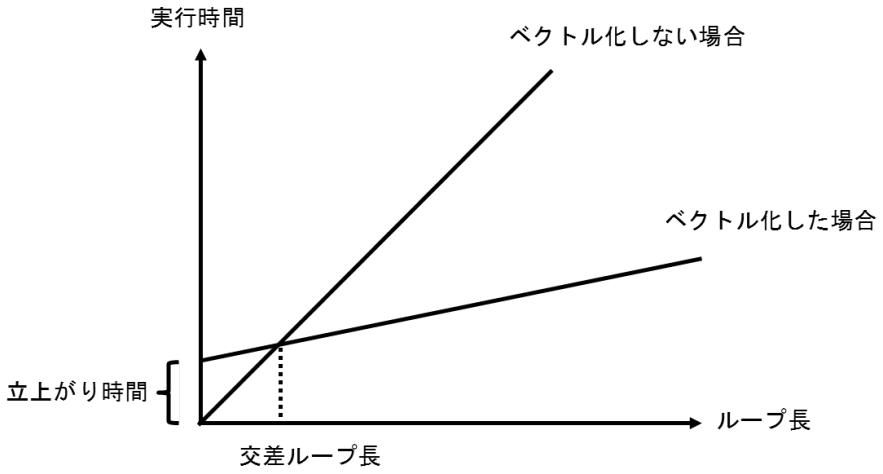

図 5.1.3-2 交差ループ長

5.2. 並列処理による高速化

5.2.1. 並列化の概要

並列処理とは, プログラム内の処理を分割し, 複数のコアを使用して同時に実行することである. 図 5.2.1-1 に並列処理の概念図を示す. 1 コアで実行したときに最も時間がかかる処理 B が並列化可能だった場合, 4 つのコアに処理を分割して並行して実行することで, プログラムの終了までの経過時間を短縮することができる. 並列処理モデルには大きく分けて「共有メモリ並列」と「分散メモリ並列」の方式がある.

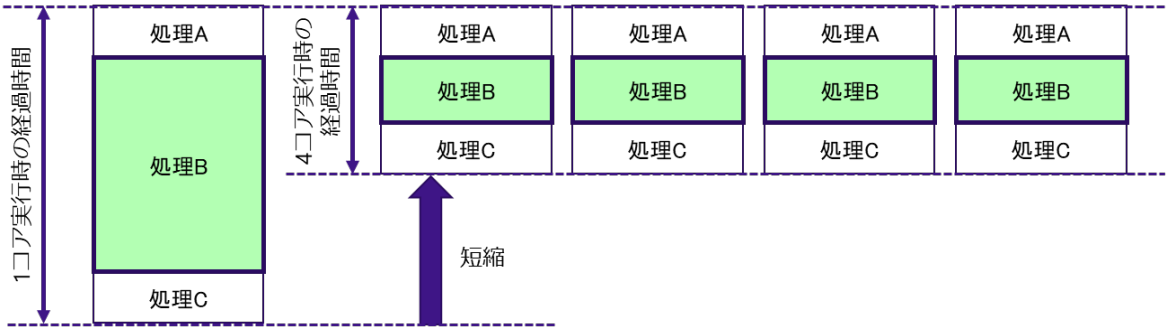

図 5.2.1-1 並列処理の概念図

5.2.2. 共有メモリ並列

共有メモリ並列とは，複数のコアが単一のメモリ空間を共有しながら並列処理を行うものである．図 5.2.2-1 に，共有メモリ並列の概念図を示す．AOBA-A では，Vector Engine（VE）ごとに共有メモリが搭載されており，8 つのベクトルコアにより共有されている．つまり，共有メモリ並列処理は VE 単位で実行され，最大並列数は8並列となる．並列化手法としては，自動並列および OpenMP 並列が利用できる．

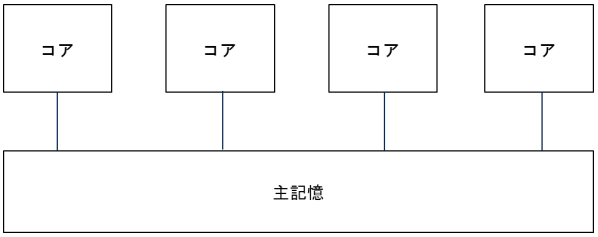

図 5.2.2-1 共有メモリ並列

(1) 自動並列

AOBA-A のコンパイラは自動並列化機能を有している．自動並列化機能を使用することで，コンパイラが自動的にプログラム内の並列実行できるループや文の集まりを抽出し，並列処理できるようにプログラムを変形する．さらに，並列処理制御のための処理の挿入などを自動的に行う．自動並列化を有効にするためには，コンパイル時に「-mparallel」オプションを指定する（Fortran プログラム/C プログラム共通）．図 5.2.2-2 に Fortran プログラムにおけるコンパイルオプションの指定例を記載する．

```
# Fortran プログラム
$ nfort -mparallel -O3 a.f90
```

図 5.2.2-2 自動並列コンパイルオプション指定例

(2) OpenMP 並列

コンパイラが自動的に並列処理を行う自動並列に対して，OpenMP 並列による並列化を行う場合，

ユーザ自身が並列化のための指示行をプログラム中に挿入する必要がある．自動並列化は，コンパイラの実装に依存するため，コンパイラ環境が変わることで並列化の挙動も変わる可能性があるが，OpenMP は標準規格であるため，異なるアーキテクチャのシステムでもソースコードを書き換える必要がなく，ユーザが意図したとおりの並列化を実現することができる．OpenMP 指示行を有効にするためには，コンパイル時に「-fopenmp」オプションを指定する（Fortran プログラム/C プログラム共通）．図 5.2.2-3 に Fortran プログラムにおけるコンパイルオプションの指定例を記載する．

```
# Fortran プログラム
$ nfort -fopenmp -O3 a.f90
```

図 5.2.2-3 OpenMP 並列コンパイルオプション指定例

図 5.2.2-4 に OpenMP 並列の基本構造を記載する．Fortran の場合，「!$OMP PARALLEL」と「!$OMP END PARALLEL」で囲まれた範囲の処理を複数のスレッドで実行する．C プログラムの場合，「#pragma omp parallel」の次のブロック（”{“ から “}” まで）の処理が並列化の対象となる．

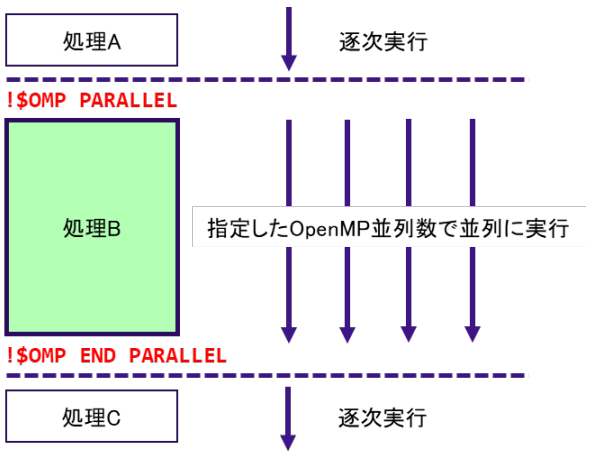

図 5.2.2-4 OpenMP 並列の基本構造

OpenMP 並列化を行う際には，並列化対象のループ内だけで定義・参照される配列や変数の取り扱いに注意する必要がある．図 5.2.2-5 の例のようなプログラムを，一番外側のループ（ループ変数 k）で OpenMP 並列化する．この場合，ループ内で使用される配列 a は k の次元を持たず，さらにループ内で定義・参照されている．もし，配列 a のメモリ領域がすべてのスレッドで共通の領域であった場合，複数のスレッドから同じ領域に対して定義（値の更新）が行われるため，そのデータを参照するタイミングによって結果が変わってしまう可能性がある．つまり，スレッドごとに別々のメモリ領域を確保しなければ本来の正しい結果を得ることはできない．このような場合には PRIVATE 宣言を行うことで，特定の配列，変数をスレッドごとに独立したメモリ領域に確保することができる．PRIVATE 宣言の指定例を図 5.2.2-6 に記載する．

```
do k = 1, nz
 do j = 1, ny
  do i = 1, nx
    a(i,j) = b(i,j,k)*c(i,k)
    d(i,j,k) = d(i,j,k) + a(i,j) * e(j,k)
  enddo
 enddo
enddo
```

図 5.2.2-5 OpenMP 化するサンプルプログラム

```
!$OMP PARALLEL DO PRIVATE(a)
do k = 1, nz
 do j = 1, ny
  do i = 1, nx
    a(i,j) = b(i,j,k)*c(i,k)
    d(i,j,k) = d(i,j,k) + a(i,j) * e(j,k)
  enddo
 enddo
enddo
!$OMP END PARALLEL DO
```

図 5.2.2-6 PRIVATE 宣言の指定例

総和演算などの処理を並列化して実行する場合にも注意が必要である. 図 5.2.2-7 に総和演算の例を記載する. この処理を並列化する場合, スレッドごとに部分和の計算結果を変数 sum に代入し, 最後に部分和同士を足し合わせて総和を求める必要がある. このとき, 変数 sum がすべてのスレッドで共通のメモリ領域だった場合, 正しく部分和を計算することができず, 正しい結果を得ることはできない. このような場合には, リダクション演算を使用する. 図 5.2.2-8 にリダクション演算の指定例を示す.

リダクション演算では, 総和演算のほかにも表 5.2.2-1 に記載されているような演算子や組み込み関数が実行可能である.

```
do i=1,n
 sum=sum+i
enddo
```

図 5.2.2-7 総和演算を行うサンプルプログラム

```
!$OMP PARALLEL DO REDUCTION(+:sum)
    do i=1,n
     sum=sum+i
    enddo
!$OMP END PARALLEL DO
```

図 5.2.2-8 OpenMP でのリダクション演算の指定例

表 5.2.2-1 リダクション演算で指定可能な演算子，組み込み関数

オペレータ	意味
+	和
*	積
-	差
.AND.	論理積
.OR.	論理和
.EQV.	論理等価
.NEQV.	論理非等価
MAX	最大
MIN	最小
IAND	ビット論理積
IOR	ビット論理和
IEOR	ビット排他的論理和

5.2.3. 分散メモリ並列

自動並列・OpenMP 並列がメモリを共有するコアによる並列化であるのに対して，分散メモリ並列はネットワークを介して接続された独立したメモリ空間をもつ複数のノード間で行うことができる並列化である．図 5.2.3-1 に，分散メモリ並列の概念図を示す．共有メモリ並列が VE 内での並列化であったのに対して，分散メモリ並列では複数の VE を使用して並列化を行うことができる．分散メモリ並列を実現するための並列プログラミング環境として MPI（Message Passing Interface）がある．MPI は分散メモリ並列計算の標準規格となっているためポータビリティに優れているが，ユーザ自身でデータや処理を分割し，プロセス間の通信の方法やタイミングをプログラミングする必要がある．

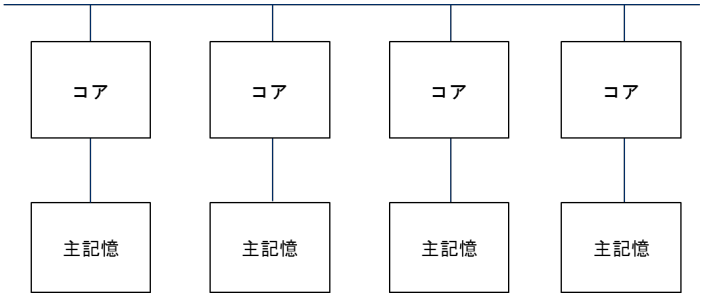

図 5.2.3-1 分散メモリ並列

(1) MPI プログラムの基本構造

図 5.2.3-2 に MPI プログラムの基本構造の概念図を記載する．MPI 化の対象となるのはプログラム内で MPI_INIT が呼び出された地点から MPI_FINALIZE が呼び出された地点までとなる．プログラム実行時の全プロセス数は MPI_COMM_SIZE により確認することができる．また，自分自身のプロセス番号は MPI_COMM_RANK により得ることができる．

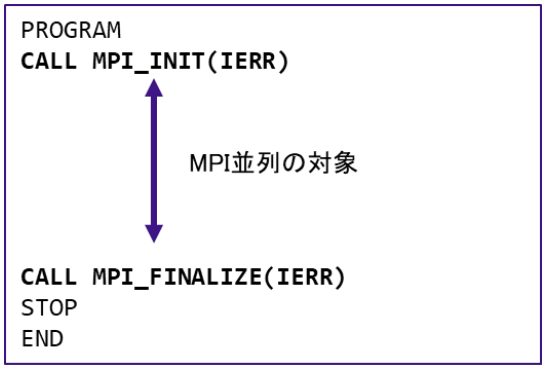

図 5.2.3-2 MPI プログラムの基本構造

(2) 1 対 1 通信

MPI では，それぞれのプロセスは独立したメモリ空間をもっているため，ほかのプロセスが更新したデータを使用して計算を行う必要がある場合，データを更新したプロセスが持っているデータを自分自身のメモリ空間にコピーして計算を行う必要がある．このとき，プロセス間通信が発生する．通信は 1 対 1 通信と集団通信に分類することができる．

図 5.2.3-3 に 1 対 1 通信の概念図を記載する．プロセス 0 のデータをプロセス 1 に転送する，プロセス 1 のデータをプロセス 2 に転送するというように，1 対 1 通信は一組の送信プロセスと受信プロセスが行うデータ転送であり，代表的なものとして MPI_SEND，MPI_RECV がある．図 5.2.3-4 は MPI_SEND と MPI_RECV の通信の動作を模式的にあらわしたものであり，プロセス 0 がもつ配列の一部をプロセス 1 に転送している．表 5.2.3-1 に代表的な 1 対 1 通信の例を記載する．

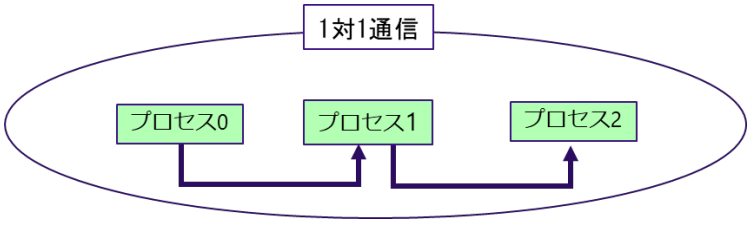

図 5.2.3-3 1 対 1 通信の概念図

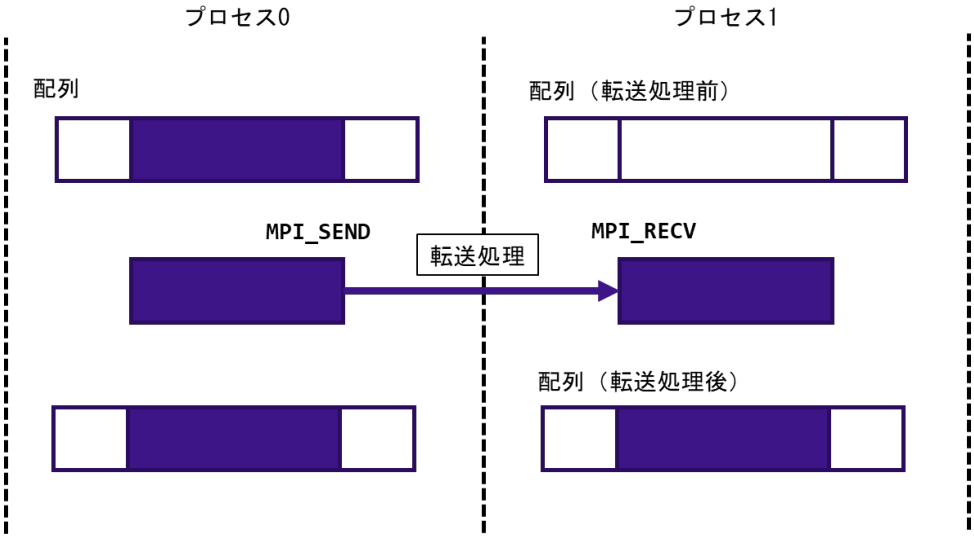

図 5.2.3-4 MPI_SEND, MPI_RECV の動作の模式図

(3) 集団通信

1 対 1 通信が特定の 1 プロセスとの間で行われるデータ転送であるのに対し，同じコミュニケータを持つ全プロセスで行う同期的通信を集団通信と呼ぶ．図 5.2.3-5 に集団通信の概念図を記載する．プロセス 1 とプロセス 2 のデータをプロセス 0 に集めるといったようにグループ内のプロセス全体でデータのやり取りを行う．代表的な集団通信として MPI_REDUCE がある．図 5.2.3-6 は MPI_REDUCE の通信の動作を模式的にあらわしたものである．MPI_REDUCE は，同じコミュニケータを持つ全プロセスが，送信バッファのデータを通信しながら指定された演算を行い，演算結果を指定されたプロセスの受信バッファに格納する．指定可能な演算として総和，最大値/最小値探索，累積などがある．この例では，最大値探索を行っており，結果をプロセス 0 の受信バッファに格納している．プロセス 0 からプロセス 3 がもつデータの中で，最大の値がプロセス 0 の受信バッファに格納されていることがわかる．集団通信には，ほかにも，代表プロセスの送信バッファのデータを全プロセスの受信バッファに送信する MPI_SCATTER や，逆に全プロセスの送信バッファから代表プロセスの受信バッファにデータを集める MPI_GATHER などもある．表 5.2.3-1 に代表的な集団通信の例を記載する．

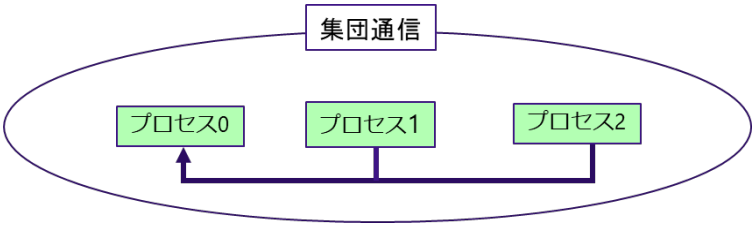

図 5.2.3-5 集団通信の概念図

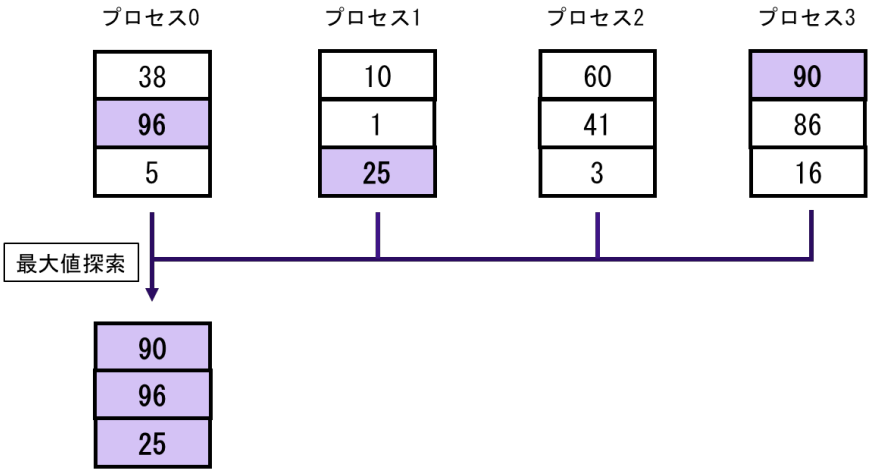

図 5.2.3-6 MPI_REDUCE(最大値探索)の動作の模式図

表 5.2.3-1 代表的な MPI 通信手続き

通信の種別	MPI 通信手続き	説明
1 対 1 通信	MPI_SEND	ブロッキング型送信
	MPI_RECV	ブロッキング型受信
	MPI_ISEND	非ブロッキング型送信
	MPI_IRECV	非ブロッキング型受信
	MPI_WAIT	通信完了の待ち合わせ
集団通信	MPI_BCAST	ブロードキャスト
	MPI_REDUCE	リダクション演算(結果を代表プロセスへ送信)
	MPI_ALLREDUCE	リダクション演算(結果を全プロセスへ送信)
	MPI_GATHER	データの集積(代表プロセスへ送信)
	MPI_ALLGATHER	データの集積(全プロセスへ送信)
	MPI_SCATTER	データの分配(代表プロセスから分配)
	MPI_ALLTOALL	全プロセスから全プロセスへのデータ分配

5.3. VH-VE 連携による高速化

第 3 章に記載のとおり, AOBA-A を構成する SX-Aurora TSUBASA は汎用的な x86 プロセッサ(AMD EPYC CPU)を搭載した VH とベクトルプロセッサを搭載した VE から構成される. このヘテロジニアスな HW 構成を活用することで, AOBA-A では 3 つのプログラミングモデルから最適なものを選択することができる. 図 5.3-1 は, AOBA-A で実現可能な 3 つのプログラミングモデルを模式的に記載したものである.

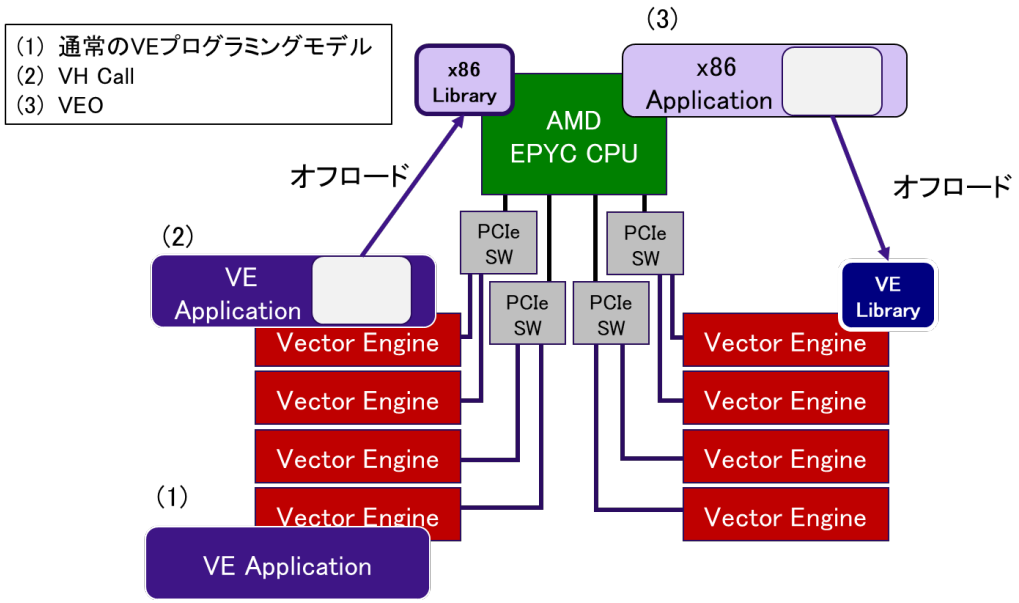

図 5.3-1 AOBA-A の実行モデル

(1)の実行モデルは，通常の VE プログラミングモデルである．ユーザアプリケーションを SX-Aurora TSUBASA 向けコンパイラでコンパイルすることで，アプリケーション全体を VE 上で実行することができる．プログラム全体を VE で実行したときに，一部の処理がベクトル化できず，その部分が性能のボトルネックとなる場合がある．そのような場合に，アプリケーションの一部を x86 プロセッサ側にオフロードして実行する実行モデルが (2) の VH Call である※1．一方，実行モデル (3) の Alternative VE Offloading (VEO) は，GPGPU などのアクセラレータで使用される実行モデルであり，x86 アプリケーションから VE カーネル関数を呼び出すことが可能である※2．このように，実行モデル(2)(3)では，VH と VE がお互いの短所を補い合うことで，より高速にアプリケーションを実行できるようになる．

また，SX-Aurora TSUBASA では，VE 上で動作する MPI プロセスと VH 上で動作する MPI プロセスを連携実行させる Scalar-Vector Hybrid(ハイブリッド実行)が可能である．図 5.3-2 にハイブリッド実行の模式図を記載する．

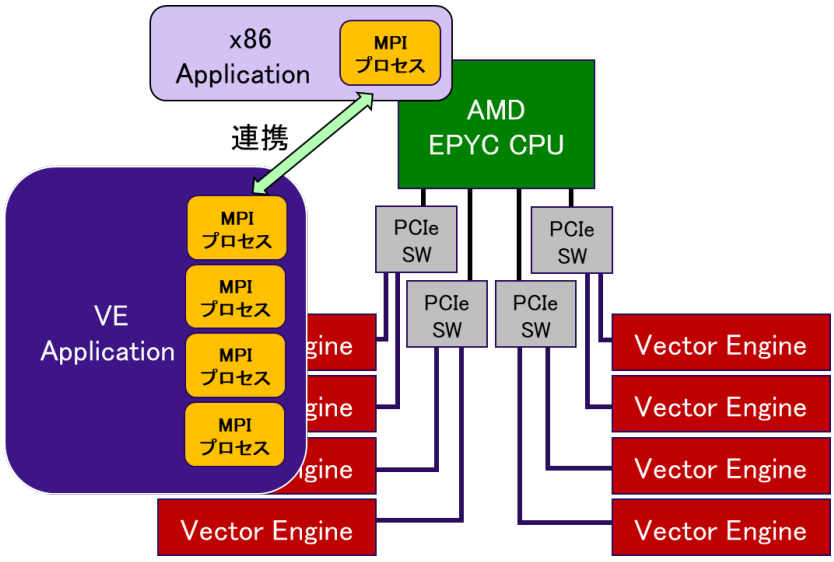

図 5.3-2 Scalar-Vector Hybrid の模式図

ハイブリッド実行が有効なケースとして，定期的に大容量の I/O 処理が発生するような場合が考えられる．VE はメモリ負荷の高い演算処理が得意である一方で，I/O 処理に関しては VH で実行したほうが高速に処理できる場合が多い．そこで，演算処理を VE の MPI プロセスで実行し，I/O 処置については VH 側の MPI プロセスが実行することで，VH と VE の両者の長所を生かすことができる．VH Call を使用することでも，演算処置と I/O 処理を VH と VE で分けて実行することは可能であるが，VH Call の場合には，VH 側の I/O 処理が終了するまで，VE 側の演算処理は待ち状態となる．一方で，ハイブリッド実行の場合には，演算処置と I/O 処理をオーバーラップして実行することができるため，より高速に実行できる．

※1 https://sxauroratsubasa.sakura.ne.jp/documents/veos/en/libsysve/md_doc_VHCall.html

※2 https://sxauroratsubasa.sakura.ne.jp/documents/veos/en/aveo/index.html

5.3.1. VH Call

VH Call により，VE が不得意とする処理を VH 側にオフロードすることができる．VH Call を利用するためには，はじめに VH 側にオフロードする処理を共有ライブラリとして作成しておく必要がある．オフロードする処理をプログラムとして切り出し，VH 用のコンパイラを使用して共有ライブラリを作成する．作成した共有ライブラリを VE 側で動作するプログラムから呼び出すことで，VH 上に処理をオフロードすることができる．VH Call では VE および VH で動作するプログラム，共有ライブラリは Fortran と C 言語の両方をサポートしている．

表 5.3.1-1 は，VE 側で動作する Fortran プログラムから VH 上で動作する共有ライブラリを呼び出す手順の概要をまとめたものである．

表 5.3.1-1 VH Call の利用手順

手順	概要	呼び出す手続き名
1	VH 上で動作する共有ライブラリを呼び出す	fvhcall_install( )
2	共有ライブラリの中から，VH 上で実行する手続きを探す	fvhcall_find( )
3	手続きに渡す引数を作成する	fvhcall_args_alloc( ) fvhcall_args_set( )
4	VE 側から手続きを呼び出す	fvhcall_invoke_with_args( )
5	引数を開放する	fvhcall_args_free( )
6	共有ライブラリをアンロードする	fvhcall_uninstall( )

まず，vhcall_install を使用して，あらかじめ作成しておいた共有ライブラリを呼び出す．次に，fvhcall_find により，呼び出した共有ライブラリに含まれる手続きの中から，今回呼び出す手続きを探す．次に，fvhcall_args_alloc により，引数のバッファを確保し，fvhcall_args_set で引数の値を設定する．VH 側の処理の呼び出しは，fvhcall_invoke_with_args により行う．この呼び出しにより，VH 上で処理が行われる．VH 側の処理が完了したら，fvhcall_args_free および fvhcall_uninstall により後処理を行う．

5.3.2. VEO

VEO により，x86 アプリケーションの処理の一部を VE 側にオフロードする．VEO を利用する場合も，最初に VE 側にオフロードする処理を切り出し，VE 用のコンパイラでコンパイルしておく必要がある．VEO は，実行ファイルや共有ライブラリに含まれる関数をサポートするため，どちらかの形式で準備しておく．VEO では，VE 上で動作する実行ファイル，共有ライブラリについては Fortran と C 言語の両方をサポートしているが，VH 上で動作するプログラムは C 言語のみがサポートされている．

表 5.3.2-1 は，VH 側で動作する C プログラムから VE 上で動作する実行ファイルもしくは共有ライブラリを呼び出す手順の概要をまとめたものである．

表 5.3.2-1 VEO の利用手順

手順	概要	呼び出す関数名
1	VE 上にプロセスを生成する	veo_proc_create( )
2	VE 上で動作する実行ファイル，共有ライブラリを呼び出す	veo_load_library( )
3	VE 上にスレッドを生成する	veo_context_open( )
4	手続きに渡す引数を作成する	veo_args_alloc( )
5	VH 側から関数を呼び出す	veo_call_async_by_name( )
6	VE 上で実行された関数の終了を待ち，戻り値を受け取る	veo_call_wait_result( )
7	引数を開放する	veo_args_free( )
8	VE 上のスレッドを開放する	veo_context_close( )
9	VE 上のプロセスを終了する	veo_proc_destroy( )

VH Call との違いは，手順 1，3 で VE 上で動作するプロセス，スレッドを生成する必要があることである．また，VEO の場合には，VE 上で実行する関数を非同期で呼び出すことが可能である．そのため手順 6 の veo_call_wait_result により VE 上で実行した関数の終了を待ち，戻り値を受け取る．

5.3.3. Scalar-Vector Hybrid

ハイブリッド実行を行うことで，VE 上で動作する MPI プロセスと VH 上で動作する MPI プロセスを連携して実行させることができる※3．MPI プロセス間の通信は，同じコミュニケータに所属するプロセス同士で行われるため，VE 上の MPI プロセスと VH 上の MPI プロセスが通信するためのコミュニケータを準備し，VE-VH 間でのデータの転送を行う．図 5.3.3-1 は，ハイブリッド実行時のコミュニケータの例を示している．

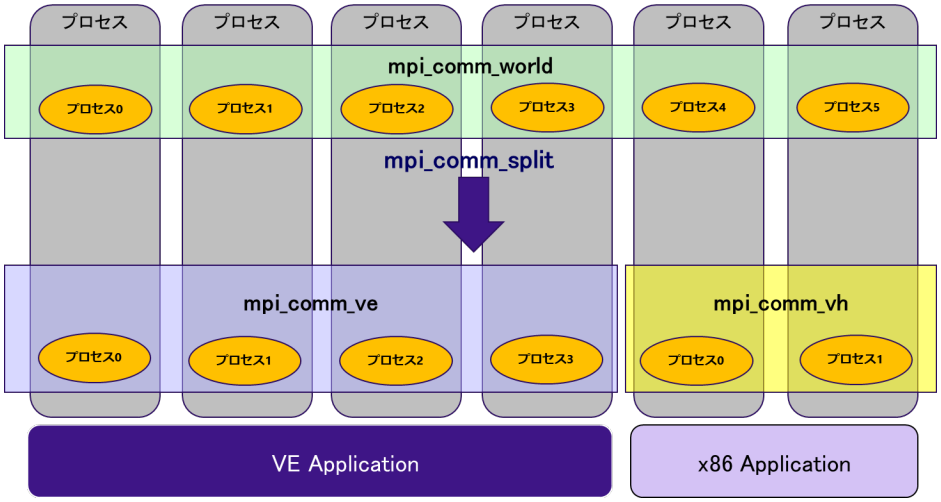

図 5.3.3-1 Scalar-Vector Hybrid のコミュニケータの分割例

この例では，VE 上の MPI プロセス間通信のためのコミュニケータ(mpi_comm_ve)と，VH 上の MPI プロセス間通信のためのコミュニケータ(mpi_comm_vh)，そして VH-VE 間の MPI プロセス間通信のためのコミュニケータ(mpi_comm_world)の合計 3 つのコミュニケータにより MPI 通信を制御する．全体で共通のコミュニケータである mpi_comm_world を mpi_comm_split により分割して新しいコミュニケータ(mpi_comm_ve および mpi_comm_vh)を生成する．

VE 向けに用意されているコンパイラコマンド(mpinfort，mpincc，mpinc++)を使用して VE application のコンパイルを行う．さらに，-vh オプションを使用することで，VE 向けのコンパイラコマンドを使用して x86 application のコンパイル(GNU コンパイラ)を行うことも可能である．図 5.3.3-2 に VE application，x86 application のコンパイル例を記載する．

```
# VE application
$ mpinfort -o ve.out ve_applicatin.f90

# x86 application
$ mpinfort -vh -o vh.out vh_application.f90
```

図 5.3.3-2 VE application および x86 application のコンパイル例

環境変数の設定により，x86 application 向けのコンパイラを GNU コンパイラから変更することができる．図 5.3.3-3 に Intel コンパイラを使用する場合の設定例を記載する．

```
$ export NMPI_CC_H=icc
$ export NMPI_CXX_H=icpc
$ export NMPI_FC_H=ifort
$ mpincc -vh vh_allication.c
$ mpinc++ -vh vh_application.cpp
$ mpinfort -vh vh_application.f90
```

図 5.3.3-3 x86 application のコンパイラの変更方法

図 5.3.3-4 にハイブリッド実行を行う場合の実行コマンドイメージを記載する.

```
$ mpirun -ve 0-1 -np 4 ./ve.out : ¥
         -vh -np 2 ./vh.out
```

図 5.3.3-4 ハイブリッド実行のコマンドイメージ

VE application では VE0 および VE1 それぞれで 2MPI プロセスが起動され, x86 application では 2MPI プロセスが起動される. 図 5.3.3-5 に, MPI プロセスの配置イメージを記載する.

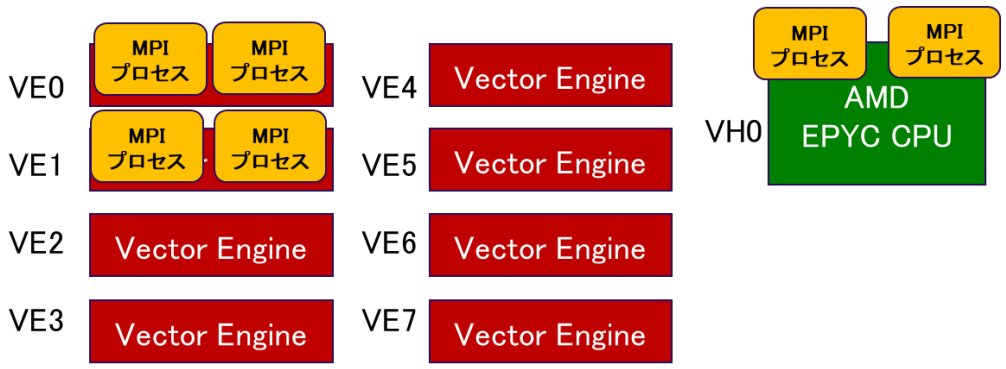

図 5.3.3-5 MPI プロセスの配置イメージ

※3 https://sxauroratsubasa.sakura.ne.jp/documents/mpi/pdfs/g2am01-NEC_MPI_User_Guide_ja.pdf

5.4. 性能解析ツール

AOBA-A では, これまでの SX シリーズと同様に様々な性能解析ツールが利用可能である. 性能解析ツールを用いた性能分析により, プログラムのベクトル演算率や平均ベクトル長といった性能指標を確認することで, 効率的にプログラムのボトルネックとなっている箇所を特定することができる. ここでは, 性能解析ツールとして診断メッセージリスト(Diagnostic Message List), 編集リスト(Format List), PROGINF, FTRACE について記載する.

5.4.1. 診断メッセージリスト

コンパイル時に -report-diagnostics, または, -report-all を指定することで「ソースファイル名.L」というファイルがカレントディレクトリに作成され, プログラムのベクトル化, 最適化, 並列化などの状況が診断メッセージとして出力される. 図 5.4.1-1 に診断メッセージリストの出力例を記載する.

```
LINE             DIAGNOSTIC MESSAGE

  11: vec( 103): Unvectorized loop.
  11: vec( 113): Overhead of loop division is too large.
  12: opt(1037): Feedback of array elements.
  12: vec( 120): Unvectorizable dependency.:
```

図 5.4.1-1 診断メッセージリストの出力例

診断メッセージリストは，プログラムの行番号と，その行に適用したベクトル化・最適化などに関する診断メッセージが出力される．この例では，11 行目からの DO ループがベクトル化されていないこと，その原因として 12 行目で計算される配列にベクトル化不可の依存関係があることがわかる．

5.4.2. 編集リスト

編集リストは，-report-format，または，-report-all を指定することで「ソースファイル名.L」ファイルに出力される．編集リストには，ループのベクトル化，並列化の情報や，手続きのインライン展開の情報が記載されており，ベクトル化されていないループを容易に特定することができる．図 5.4.2-1 に編集リストの出力例を記載する．オリジナルのソースコードの情報に，それぞれのループのベクトル化や並列化の状況などが視覚的にわかりやすく追記されている．

```
LINE   LOOP      STATEMENT

           :
   10:
   11: +------>  do i = 1, n
   12: |          c(i) = a(i) * c(i-1) + b(i) * c(i-2)
   13: +------   enddo
   14:
           :
```

図 5.4.2-1 編集リストの出力例

この例では，11 行目から 13 行目の DO ループに ”+” 印が記載されている．これは，この DO ループがベクトル化されていないことを示している．

以下に，ループのベクトル化，並列化，インライン展開に関する情報の出力例を記載する．

(1) ループ全体がベクトル化された場合

ベクトル化されたループに ”V” が表示される．

```
V------>          DO I=1,N
|
V------           END DO
```

図 5.4.2-2 ループ全体がベクトル化された場合の編集リスト

(2) ループが部分ベクトル化された場合

部分ベクトル化されたループに“S”が表示される.

```
S------>          DO I=1,N
|
S------           END DO
```

図 5.4.2-3 ループが部分ベクトル化された場合の編集リスト

(3) ループが条件ベクトル化された場合

条件ベクトル化されたループに“C”が表示される.

```
C------>          DO I=1,N
|
C------           END DO
```

図 5.4.2-4 ループが条件ベクトル化された場合の編集リスト

(4) ループが並列化された場合

並列化されたループに“P”が表示される.

```
P------>          DO I=1,N
|
P------           END DO
```

図 5.4.2-5 ループが並列化された場合の編集リスト

(5) ループが並列化, かつ, ベクトル化された場合

並列化かつベクトル化されたループに“Y”が表示される.

```
Y------>          DO I=1,N
|
Y------           END DO
```

図 5.4.2-6 ループが並列化かつベクトル化された場合の編集リスト

(6) ループがベクトル化されなかった場合

ベクトル化されなかったループには“+”が表示される.

```
+------>          DO I=1,N
|
+------           END DO
```

図 5.4.2-7 ループがベクトル化されなかった場合の編集リスト

(7) 配列式など, 1 行にループ全体が含まれる場合

ループの構造は“=”で表示される. この例では配列式がベクトル化されていることを示す.

```
V======>          A = A + B
```

図 5.4.2-8 配列式など, 1 行にループ全体が含まれる場合の編集リスト

(8) ループが一重化された場合

一重化されたループの外側ループに“W”, 内側ループに“*”が表示される.

```
W------>        DO I=1,N
|*----->          DO J=1,M
||
|*-----           END DO
W------         END DO
```

図 5.4.2-9 ループが一重化された場合の編集リスト

(9) ループが入れ換えられ, ベクトル化された場合

入れ換えた結果ベクトル化されるループに“X”が表示される.

```
X------>        DO I=1,N
|*----->          DO J=1,M
||
|*-----           END DO
X------         END DO
```

図 5.4.2-10 ループが入れ換えられ, ベクトル化された場合の編集リスト

(10) 外側ループがループアンロールされ, 内側ループがベクトル化された場合

アンロールされるループに“U”が表示され, ベクトル化されるループに“V”が表示される.

```
U------>        DO I=1,N
|V----->          DO J=1
||
|V-----           END DO
U------         END DO
```

図 5.4.2-11 外側ループがループアンロールされ, 内側ループがベクトル化された場合の編集リスト

(11) ループが融合された場合

融合したループの範囲について, 記号が表示される.

```
V------>        DO I=1,N
|
|                END DO
|                DO I=1,N
|
V------         END DO
```

図 5.4.2-12 ループが融合された場合の編集リスト

(12) ループが展開された場合

展開されるループに“*”が表示される.

```
*------>        DO I=1,4
|
*------         END DO
```

図 5.4.2-13 ループが展開された場合の編集リスト

また，ループ処理ではなく，その行の処理がどのように最適化されたかは，その行に付加された文字により確認できる．

- “I” 関数呼び出しがインライン展開された
- “M” この行を含む多重ループがベクトル行列積ライブラリ呼び出しに置き換えられた
- “F” 式に対してベクトル積和命令が生成された
- “R” 配列に retain 指示行が適用された
- “G” ベクトル収集命令が生成された
- “C”ベクトル拡散命令が生成された
- “V”配列に vreg 指示行※1，または，pvreg 指示行※1 が適用された

※1 https://sxauroratsubasa.sakura.ne.jp/documents/sdk/pdfs/g2af02-FortranUsersGuide-033.pdf

5.4.3. PROGINF

PROGINF は，プログラム全体の性能解析情報を出力する機能であり，プログラム終了時に，標準エラー出力ファイルに出力される．PROGINF を確認することで，プログラム全体のベクトル演算率，平均ベクトル長といった性能指標を確認することができる．図 5.4.3-1 に PROGINF の出力例を記載する．

```
          ********  Program  Information  ********
①  Real Time (sec)                          :            33.060014
②  User Time (sec)                          :           109.886525
③  Vector Time (sec)                        :           104.243699
④  Inst. Count                             :         77145324923
⑤  V. Inst. Count                          :         16268966755
⑥  V. Element Count                        :       4164804790539
⑦  V. Load Element Count                   :       1921282387058
⑧  FLOP Count                              :       1281280040326
⑨  MOPS                                    :          42021.575083
⑩  MOPS (Real)                             :         137499.248243
⑪  MFLOPS                                  :          11844.518710
⑫  MFLOPS (Real)                           :          38756.577193
⑬  A. V. Length                            :            255.996884
⑭  V. Op. Ratio (%)                        :             98.660787
⑮  L1 Cache Miss (sec)                     :              0.414586
⑯  CPU Port Conf. (sec)                    :              0.000000
⑰  V. Arith. Exec. (sec)                   :             35.594797
⑱  V. Load Exec. (sec)                     :             68.634671
⑲  VLD LLC Hit Element Ratio (%)           :              0.001852
⑳  FMA Element Count                       :        2002000063
㉑  Power Throttling (sec)                  :              0.000000
㉒  Thermal Throttling (sec)                :              0.000000
㉓  Memory Size Used (MB)                   :           7884.000000
㉔  Non Swappable Memory Size Used (MB      :            179.000000

㉕  Start Time (date)             :       Thu Jun  1 04:43:34 2023 JST
㉖  End Time (date)               :       Thu Jun  1 04:44:08 2023 JST
```

図 5.4.3-1 PROGINF の出力例

表示される各項目の意味は以下の通り.

① Real Time	経過時間(秒)
② User Time	ユーザ時間(秒)
③ Vector Time	ベクトル命令実行時間(秒)
④ Inst. Count	全命令実行数
⑤ V. Inst. Count	ベクトル命令実行数
⑥ V. Element Count	ベクトル命令実行要素数
⑦ V. Load Element Count	ベクトル命令ロード要素数
⑧ FLOP Count	浮動小数点データ実行要素数
⑨ MOPS	ユーザ時間 1 秒あたりに実行された演算数(100 万単位)
⑩ MOPS (Real)	経過時間 1 秒あたりに実行された演算数(100 万単位)
⑪ MFLOPS	ユーザ時間 1 秒あたりに処理された浮動小数点データ実行要素数(100 万単位)
⑫ MFLOPS (Real)	経過時間 1 秒あたりに処理された浮動小数点データ実行要素数(100 万単位)
⑬ A. V. Length	平均ベクトル長
⑭ V. OP. RATIO	ベクトル演算率(%)
⑮ L1 Cache Miss	L1 キャッシュミス時間※1
⑯ CPU Port Conf	CPU ポート競合時間※1
⑰ V. Arith Exec	ベクトル演算実行時間※1
⑱ V. Load Exec	ベクトルロード実行時間※1 (Vector Time - V.Arith Exec - V.Load Exec がベクトルストア実行時間やレジスタ間転送などの転送命令実行時間となる)
⑲ VLD LLC Hit Element Ratio	ベクトルロード命令によりロードされた要素のうち, LLC からロードされた要素の比率※1
⑳ FMA Element Count	FMA 命令実行要素数※1
㉑ Power Throttling	電力要因による HW 停止時間※1
㉒ Thermal Throttling	温度要因による HW 停止時間※1
㉓ Memory Size	メモリの最大使用量(メガバイト)
㉔ Non Swappable Memory Size Used	Partial Process Swapping 機能※2 でスワップアウトできないメモリの最大使用量
㉕ Start Time (date)	プログラムの実行開始時間
㉖ End Time (date)	プログラムの実行終了時間

上記のうち※1 の項目は, VE_PROGINF 環境変数の設定に DETAIL を指定したときに出力される.

また，VE_PERF_MODE 環境変数の設定に VECTOR-MEM を指定することで，主にメモリアクセスに関する情報を出力させることができる．図 5.4.3-2 に VECTOR-MEM を指定した場合の出力例を記載する．なお，VE_PERF_MODE の指定が変わると，メモリアクセス数の計算方法が変わるため，1 回の情報採取ではどちらか一方の性能情報しか表示させることしかできない．

※2 https://sxauroratsubasa.sakura.ne.jp/documents/nqsv/pdfs/g2ad06-NQSVUG-JobManipulator.pdf

```
          ********  Program  Information  ********
    Real Time (sec)                          :          41.899895
    User Time (sec)                          :         167.316072
    Vector Time (sec)                        :          92.309990
    Inst. Count                              :       175930159073
    V. Inst. Count                           :        16278237491
    V. Element Count                         :       564896066618
    V. Load Element Count                    :       167991674972
    FLOP Count                               :       432838957109
    MOPS                                     :         4956.779900
    MOPS (Real)                              :        19821.081454
    MFLOPS                                   :         2583.413548
    MFLOPS (Real)                            :        10330.507186
    A. V. Length                             :           34.702533
    V. Op. Ratio (%)                         :           80.776073
①   L1 I-Cache Miss (sec)                    :            0.256811
②   L1 O-Cache Miss (sec)                    :           32.683728
③   L2 Cache Miss (sec)                      :           32.884266
    FMA Element Count                        :       105937492420
④   Required B/F                             :            5.773960
⑤   Required Store B/F                       :            2.195626
⑥   Required Load B/F                        :            3.578334
⑦   Actual V. Load B/F                       :            0.404243
    Power Throttling (sec)                   :            0.000000
    Thermal Throttling (sec)                 :            0.000000
    Memory Size Used (MB)                    :         1028.000000
    Non Swappable Memory Size Used (MB)      :          179.000000

    Start Time (date)             :          Thu Jun  1 17:03:50 2023 JST
    End Time (date)               :          Thu Jun  1 17:04:32 2023 JST
```

図 5.4.3-2 PROGINF の出力例（VECTOR-MEM 指定時）

表示される各項目の意味は以下の通り．

① L1 I-Cache Miss	L1 命令キャッシュミス時間（秒）
② L1 O-Cache Miss	L1 オペランドキャッシュミス時間（秒）
③ L2 Cache Miss	L2 キャッシュミス時間（秒）
④ Required B/F	ロード命令とストア命令に指定されたバイト数から算出した B/F
⑤ Required Store B/F	ストア命令に指定されたバイト数から算出した B/F
⑥ Required Load B/F	ロード命令に指定されたバイト数から算出した B/F
⑦ Actual V. Load B/F	ベクトルロード命令により実際に発生したメモリアクセスのバイト数から算出した B/F

5.4.4. FTRACE

PROGINF を使用することで，プログラム全体の性能情報を採取することが可能であるが，プログラムの中で性能のボトルネックとなる箇所を特定するためには，より細かい単位（手続きごと）での性能情報の採取が必要になる．FTRACE を使用することで，手続き（サブルーチン）ごとや，ユーザの指定した任意の区間の性能情報（実行時間，ベクトル演算率，平均ベクトル長など）を採取することができる．プログラムの実行に時間がかかっている部分をチューニングすることで，効率よくプログラムを高速化することができる．FTRACE を使用するためには，コンパイル時に「-ftrace」オプションを指定する．プログラム実行後，実行ディレクトリに ftrace.out というファイルが作成される．この ftrace.out が存在するディレクトリで ftrace コマンドを実行することで手続きごとの性能情報が出力される．図 5.4.4-1 に FTRACE による性能解析情報の例を記載する．

```
    ①         ②              ③         ④       ⑤      ⑥    ⑦       ⑧       ⑨       ⑩       ⑪       ⑫
FREQUENCY  EXCLUSIVE        AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
           TIME[sec](  % )    [msec]                      RATIO V.LEN     TIME    MISS     CONF HIT E.%
     1012    49.093( 24.0)     48.511  23317.2  14001.4  96.97  83.2    42.132   5.511    0.000   80.32 funcA
   160640    37.475( 18.3)      0.233  17874.6   9985.9  95.22  52.2    34.223   1.973    2.166   96.84 funcB
      :
-------------------------------------------------------------------------------------------------------------
 54851346   204.569(100.0)      0.004  22508.5  12210.7  95.64  76.5   154.524  17.740   13.916   90.29     total
```

図 5.4.4-1 FTRACE の出力例

表示される各項目の意味は以下の通り．

① FREQUENCY	手続きの呼び出し回数
② EXCLUSIVE TIME	手続きの実行に要した EXCLUSIVE な CPU 時間（秒）と，全手続きの CPU 時間に関する比率（%）
③ AVER.TIME	手続きの 1 回の実行に要した EXCLUSIVE な CPU 時間の平均（ミリ秒）
④ MOPS	“EXCLUSIVE TIME” 1 秒あたりに実行された演算数（100 万単位）
⑤ MFLOPS	“EXCLUSIVE TIME” 1 秒あたりに処理された浮動小数点データ実行要素数（100 万単位）
⑥ V.OP RATIO	ベクトル演算率（%）
⑦ AVER V.LEN	平均ベクトル長
⑧ VECTOR TIME	ベクトル命令実行時間（秒）
⑨ L1CACHE MISS	L1 キャッシュミス時間（秒）
⑩ CPU PORT CONF	CPU ポート競合時間（秒）
⑪ VLD LLC HIT E. %	ベクトルロード命令によりロードされた要素のうち，LLC からロードされた要素の比率
⑫ PROC.NAME	手続き名

PROGINF と同様に，FTRACE でも VE_PERF_MODE 環境変数の設定に VECTOR-MEM を指定することで，主にメモリアクセスに関する情報を出力させることができる．図 5.4.4-2 に VECTOR-MEM を指定した場合の出力例を記載する．

```
                                        ①          ②          ③       ④       ⑤       ⑥       ⑦          ⑧           ⑨
FREQUENCY  EXCLUSIVE          ..  L1ICACHE   L1OCACHE   L2CACHE   REQ.    REQ.ST  REQ.LD  ACT.VLD   FLOP COUNT   FMA ELEM.  PROC.NAME
           TIME[sec](  %  )   ..      MISS       MISS       MISS    B/F     B/F     B/F     B/F
     15562   17.311( 43.4)    ..     0.005      4.441      4.442   5.12    1.34    3.78    0.10  16689871512  4486524600  funcA
     15562   17.235( 43.2)    ..     0.009      4.007      4.009   5.12    1.34    3.78    0.01  16689871512  4187422960  funcB
       :
------------------------    ..     ------------------------------------------------------------------------------------------
   3272238   39.862(100.0)    ..     0.038      8.620      8.620   6.26    2.29    3.97    0.30 103262749091 25237042300  total
```

図 5.4.4-2 FTRACE の出力例(VECTOR-MEM 指定時)

表示される各項目の意味は以下の通り．

① L1ICACHE MISS	L1 命令キャッシュミス時間(秒)
② L1OCACHE MISS	L1 オペランドキャッシュミス時間(秒)
③ L2CACHE MISS	L2 キャッシュミス時間(秒)
④ REQ. B/F	ロード命令とストア命令に指定されたバイト数から算出した B/F
⑤ REQ. ST B/F	ストア命令に指定されたバイト数から算出した B/F
⑥ REQ. LD B/F	ロード命令に指定されたバイト数から算出した B/F
⑦ ACT. VLD B/F	ベクトルロード命令により実際に発生したメモリアクセスのバイト数から算出した B/F
⑧ FLOP COUNT	浮動小数点データ実行要素数
⑨ FMA ELEM.	FMA 命令実行要素数

MPI プログラムについても FTRACE 情報を採取することができる．MPI プログラムの場合には，上記の情報に加えて，MPI 通信情報が含まれる．図 5.4.4-3 に MPI 通信情報の出力例を示す．

```
    ①           ②           ③           ④           ⑤           ⑥          ⑦          ⑧
 ELAPSED     COMM.TIME   COMM.TIME   IDLE TIME   IDLE TIME   AVER.LEN    COUNT    TOTAL LEN  PROC.NAME
 TIME[sec]     [sec]     / ELAPSED     [sec]     / ELAPSED    [byte]               [byte]

   369.331       6.046       0.016       3.588       0.010      45.6K    193991       8.4G   funcA
   328.229       0.000       0.000       0.000       0.000       0.0         0       0.0    funcB
   217.655      68.771       0.316      65.908       0.303      14.0K    550020       7.3G   funcC
   172.524      64.696       0.375      32.005       0.186       2.2K   4428720       9.5G   funcD
   140.347       0.000       0.000       0.000       0.000       0.0         0       0.0    funcE
```

図 5.4.4-3 MPI プログラムにおける FTRACE の出力例

表示される各項目の意味は以下の通り．

① ELAPSED TIME	経過時間(秒)
② COMM.TIME	MPI 手続きの実行に費やした経過時間(秒)
③ COMM.TIME / ELAPSED	各プロセスにおいて，MPI 手続きの実行に費やした時間経過時間が，経過時間全体に占める割合

④ IDLE TIME	メッセージ待ちに費やした経過時間
⑤ IDLE TIME / ELAPSED	各プロセスにおいて，メッセージ待ちに費やした経過時間が，経過時間全体に占める割合
⑥ AVER.LEN	MPI 手続きあたりの平均通信量(バイト，単位は 1024 換算)
⑦ COUNT	MPI 手続きによる転送回数
⑧ TOTAL LEN	MPI 手続きによる総通信量(バイト，単位は 1024 換算)

FTRACE を使用することで詳細な性能情報を採取することが可能になるが，性能情報採取のためのタイマールーチンがプログラム中に埋め込まれるため実行時間が長くなる場合がある．そのため，FTRACE はあくまで性能分析のときにのみ使用し，それ以外のケースでは「-ftrace」オプションは無効にする必要がある．

5.4.5. FILEINF

プログラム実行時に，環境変数 VE_FORT_FILEINF に YES もしくは DETAIL を指定することで，ファイル入出力情報を出力させることができる．図 5.4.5-1 に VE_FORT_FILEINF に DETAIL を指定した場合の出力例を示す．

```
          ****** File Information ******
①  Unit No . : 10
②  File Name : fort.10
    Named     : YES
③  Current Directory  : /usr/uhome/xxxxxxxx
    TMPDIR            : /tmp


    I/O Exec. Count    :      READ      WRITE       OPEN      CLOSE    INQUIRE
                              1          1          0          1          0
                          REWIND   BACKSPACE  ENDFILE
                              1          0          0
                            WAIT      FLUSH
                              0          0



Format              :       FORMATTED    Access              :     SEQUENTIAL
Blank(OPEN)         :            NULL    Blank(READ)         :           NULL
Delim(OPEN)         :            NONE    Delim(WRITE)        :           ----
Pad(OPEN)           :             YES    Pad(READ)           :            YES
Decimal(OPEN)       :           POINT    Decimal(R/W)        :          POINT
Sign(OPEN)          :       PROCESSOR    Sign(WRITE)         :      PROCESSOR
Round(OPEN)         :       PROCESSOR    Round(R/W)          :      PROCESSOR
Asynchronous        :              NO    Encoding            :       DEFAULT
Position            :          REWIND
Recl (Byte)         :           65536
File Size (Byte)    :              13    File Descriptor     :             5
File System Type    : NFS(0x00006969)    Open Mode           :     READWRITE
Terminal Assignment : NO                 Shrunk File         :           YES
Max File Size(Byte) : 600
```

```
I/O Buffer Size (KByte)   :        512
Record Buffer Size (Byte) :      65536



                          Total(In/Out)            Input            Output
④    Total Data Size (Byte)    :          25,             13,              12
⑤    Max   Data Size (Byte)    :                          13,              12
⑥    Min   Data Size (Byte)    :                          13,              12
⑦    Ave   Data Size (Byte)    :          12,             13,              12
⑧    Transfer Rate (KByte/sec) :      18.789,          19.261,          18.303


                          Total(In/Out/Aux)        Input            Output
Real Time (sec)           :    0.004284,       0.000659,        0.000640
User Time (sec)           :    0.002874,       0.000062,        0.000129



Environment Variable List :
```

図 5.4.5-1 FILEINF の出力例(DETAIL 指定時)

表示される各項目のうち主なものの意味は以下の通り.

① Unit No.	外部装置識別子
② File Name	FILE 指定子あるいは事前接続の際に指定した名前
③ Current Directory	現在作業しているディレクトリ名
④ Total Data Size	入出力を行った総転送量を、入出力の合計、入力の合計、出力の合計の順にバイト単位で出力します。
⑤ Max Data Size	入出力の中で最も大きい記録長の値を入力，出力の順にバイト単位で出力
⑥ Min Data Size	入出力の中で最も小さい記録長の値を入力，出力の順にバイト単位で出力
⑦ Ave Data Size	平均記録長を，入出力の合計，入力の合計，出力の合計の順にバイト単位で出力
⑧ Transfer Rate	ファイル転送速度を秒あたりのキロバイト(1024 バイト)単位で示したもの. 転送速度は，Total Data Size を Real Time で割った値をキロバイト(1024 バイト)単位で示した値で出力

上記以外のパラメータの意味については以下を参照のこと.

https://sxauroratsubasa.sakura.ne.jp/documents/sdk/pdfs/g2af02-FortranUsersGuide-033.pdf

6. 高速化事例

スーパーコンピューティング研究部　滝沢寛之　高橋慧智　下村陽一
情報部デジタルサービス支援課　大泉健治　小野敏　山下毅　齋藤敦子　森谷友映
高性能計算技術開発(NEC)共同研究部門　撫佐昭裕　磯部洋子　曽我隆　山口健太
日本電気株式会社　加藤季広

本章では，最適化方法およびその最適化の効果について具体的な事例を用いて説明する．表 6-1 に，本章で説明する最適化事例の一覧を記載する．

表 6-1 最適化事例一覧

事例		どのような場合に適用するか
6.1.1	ループ展開による高速化(1)	コンパイラが自動でループ展開できない場合
6.1.2	ループ展開による高速化(2)	
6.1.3	ループの一重化による高速化(1)	ベクトル化対象のループ長が短い場合
6.1.4	ループの一重化による高速化(2)	
6.1.5	ループ入れ替えによる高速化(1)	ベクトル化対象のループ長が短い場合
6.1.6	ループ入れ替えによる高速化(2)	
6.1.7	ループブロッキングによる高速化	ベクトル化対象のループ長が短い場合
6.1.8	ループ分割による高速化(1)	ベクトル化不可とする処理を含む場合
6.1.9	ループ分割による高速化(2)	
6.1.10	依存関係解消による高速化(1)	間接参照によるベクトル化不可の依存関係がある場合
6.1.11	依存関係解消による高速化(2)	
6.1.12	インライン展開による高速化	呼び出し回数の多いサブルーチンがある場合
6.1.13	ライブラリ利用による高速化(1)	乱数生成ライブラリを利用する場合
6.1.14	ライブラリ利用による高速化(2)	1 次元多重 FFT 関数を利用する場合
6.2.1	メモリアクセスパターンの変更による高速化	間接アクセスによりメモリ負荷が高い場合
6.2.2	メモリアクセス効率化による高速化	多重ループのループ長がすべて同じでどのループでベクトル化すべきか不明な場合
6.2.3	ループアンロールによる高速化	アンロールすることでメモリ負荷を低減できる場合
6.2.4	メモリ割当回数低減による高速化	メモリ割当回数が低減できる場合
6.3.1	ファイル I/O の高速化	ファイル I/O を高速化する場合
6.4.1	複数の通信処理をまとめることによる高速化	通信処理のコストが高い場合
6.4.2	演算と通信のオーバーラップによる高速化	
6.5.1	VH Call による高速化(1)	ベクトル性能の低い処理が高コストの場合
6.5.2	VH Call による高速化(2)	
6.5.3	VH-VE Hybrid MPI による高速化	高コストの I/O 処理が頻繁に発生する場合

6.1. ベクトル化推進の事例

6.1.1. ループ展開による高速化(1)

(1) 最適化方針

多重 DO ループがある場合, コンパイラは原則的に最内ループに対してベクトル化を行う. 本事例は, ベクトル化を行った最内ループのベクトル長が短く性能が出ていないため, 最内ループを展開して高速化を行った例である.

図 6.1.1-1 に最適化前のコードを示す. 四重DOループの最内ループは, コンパイラの自動ベクトル化機能によりベクトル化されている. 図 6.1.1-2 に示した最適化前の FTRACE 情報では, 平均ベクトル長が 14.4 と低い値になっている. これは, ベクトル化されたインデックス L を持つ最内ループの繰り返し数が 13 と短いためである. 最内ループを展開する UNROLL 指示行を挿入し, コンパイラでループを展開して, その外側のループ長の長い DO ループをベクトル化することによって実行時間の短縮を図る(注 1).

```
! 最適化前

+------>   do k=kst,ked
|+----->      do j=1,jn
||V---->        do i=1,in
|||V--->          do l=1,13          <-- ループ長 13
||||                 a(l) = a(l) &
||||                    +(var1*b(i,j,k)*c(i,j,k)*d(e(i,j,k))*(var2**3))*f(i,j,k,l)
||||                 g(l) = g(l) +(var1*c(i,j,k)*d(e(i,j,k))*(var2**3))*f(i,j,k,l)
|||V---            enddo
||V----          enddo
|+-----       enddo
+------     enddo
```

図 6.1.1-1 ループ展開前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
1800	270.342(81.1)	150.190	5361.8	2177.3	79.88	**14.4**	211.370	2.514	87.272	84.12	【最適化前】

図 6.1.1-2 ループ展開前の FTRACE 情報

(注 1)コンパイラによる最適化レベルが 2 (-O2)以上である場合, 最内ループ長がコンパイル時に確定しており, それをコンパイラが検出可能であれば, ループ長が 4 以下の場合はそのループが自動的に展開される. 自動ループ展開の対象とするか否かの閾値 m(既定値は m=4)はコンパイラオプション-floop-unroll-complete=m で指定できる. 既定値が m=4 と比較的小さな値となっているのは, 自動ループ展開適用の結果としてコンパイルに要する時間が極端に増加したり, コード行数の増加によってレジスタスピルが発生して性能が低下したりする懸念があるためである.

(2) 最適化内容

図 6.1.1-3 に最適化後のコードを示す. 最内ループのループ長については, コード上に即値で記述されているので, UNROLL 指示行にループ長 13 を指定して最内ループを展開する. 編集

リストでは最内ループが展開されたことを示す，＊マークが付加され，その外側の DO ループにベクトル化されたことを示す V マークが付加されている．

```
! 最適化後

+------>   do k=kst,ked
|+----->      do j=1,jn
||V---->         do i=1,in
|||       !$NEC unroll(13)        <-- 指示行によるループ展開
|||*--->            do l=1,13
||||    G              a(l) = a(l) &
||||                      +(var1*b(i,j,k)*c(i,j,k)*d(e(i,j,k))*(var2**3))*f(i,j,k,l)
||||                   g(l) = g(l) +(var1*c(i,j,k)*d(e(i,j,k))*(var2**3))*f(i,j,k,l)
|||*---             enddo
||V----           enddo
|+-----        enddo
+------     enddo
```

図 6.1.1-3 ループ展開後のコード

(3) 性能分析

図 6.1.1-4 に最適化前後の性能値を示す．

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
1800	**270.342**(81.1)	150.190	5361.8	2177.3	79.88	**14.4**	211.370	2.514	87.272	84.12	【最適化前】
1800	**21.813**(26.0)	12.118	69390.0	30512.9	99.51	**198.1**	21.808	0.004	0.000	37.94	【最適化後】

図 6.1.1-4 ループ展開前後の FTRACE 情報

UNROLL 指示行の挿入により，最内ループが展開され，その外側の DO ループがベクトル化されたことで，平均ベクトル長が 14.4 から 198.1 に増加し，実行時間が 270.4 秒から 21.9 秒に短縮されている．

6.1.2. ループ展開による高速化(2)

(1) 最適化方針

本事例は最内ループの繰り返し範囲が配列もしくは変数で定義され，コンパイル時に繰り返し数を確定することが出来ないため，ループ展開の対象とならない DO ループに対し，ユーザが DO ループの繰り返し範囲を定数として明記することで，ループ展開の対象とする手法である．

図 6.1.2-1 に最適化前のコードを示す．本コードは三重ループの最内 DO ループの終値が配列 jn1(k)-1 または変数 jn2 で与えられているが，プログラムの実行中に取り得る値は，jn1(k)-1=2 および，jn2=5 と定数になることが明確になっている．

このことから，DO ループの終値を定数で記述することでコンパイラのループ展開の対象とし，その外側のループ長の大きい DO ループでベクトル化することにより実行時間の短縮を図る．

図 6.1.2-2 に示した FTRACE 情報から，最適化前はベクトル演算率が 42.54% ，平均ベクトル長が 4.8 と低く，ベクトルプロセッサの性能を十分に発揮できていないことが分かる．

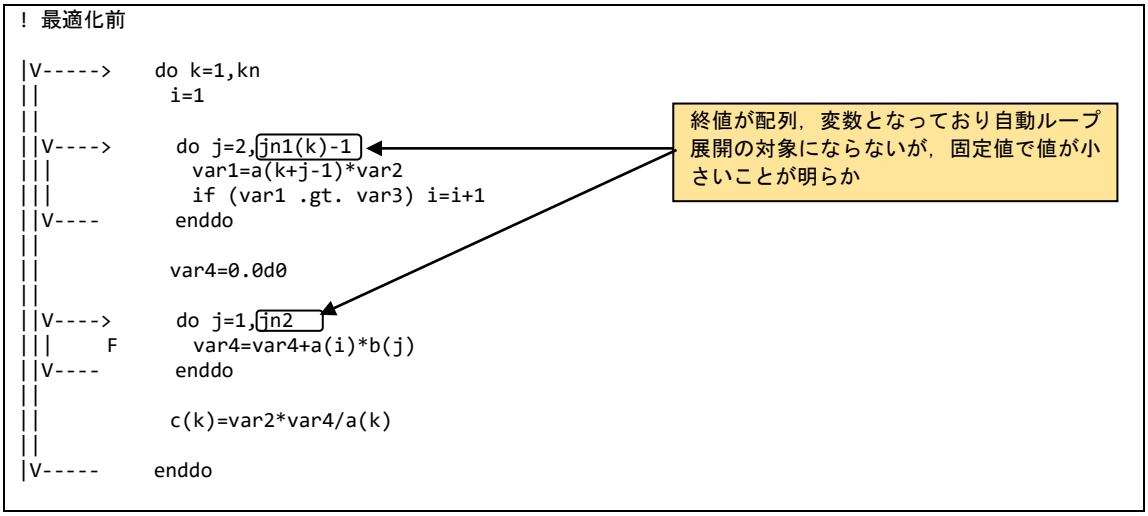

図 6.1.2-1 ループ展開前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
10000	3.234(100.0)	0.323	668.4	196.8	**42.54**	**4.8**	3.232	0.001	0.000	100.00	【最適化前】

図 6.1.2-2 ループ展開前の FTRACE 情報

(2) 最適化内容

図 6.1.2-3 に最適化後のコードを示す. 最内ループの終値を定数で指定することで, コンパイラにより最内 DO ループのループ展開が自動で行われ, その外側の DO ループがベクトル化されたことが確認できる.

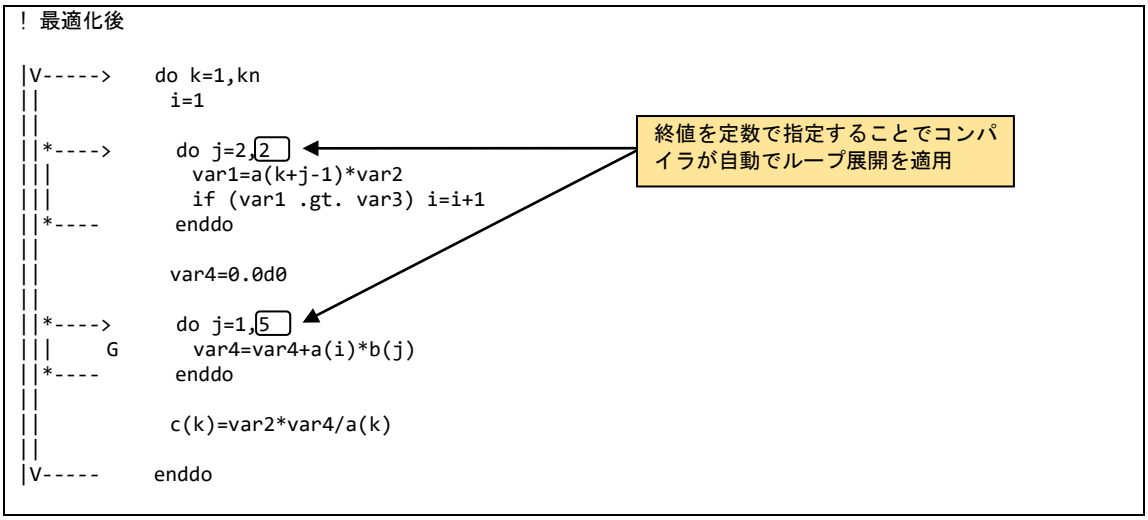

図 6.1.2-3 ループ展開後のコード

(3) 性能分析

図 6.1.2-4 に，最適化前後の性能値を示す．

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
10000	**3.234**(100.0)	0.323	668.4	196.8	**42.54**	**4.8**	3.232	0.001	0.000	100.00	【最適化前】
10000	**0.002**(69.0)	0.000	328481.9	251176.4	**99.41**	**252.6**	0.004	0.000	0.000	100.00	【最適化後】

図 6.1.2-4 ループ展開前後の FTRACE 情報

最内ループが展開され，その外側の DO ループがベクトル化されたことにより，ベクトル演算率が 42.54% から 99.41 %に，平均ベクトル長が 4.8 から 252.6 まで増加し，実行時間が 3.3 秒から 0.002 秒に短縮されている．

6.1.3. ループの一重化による高速化(1)

(1) 最適化方針

多重 DO ループがある場合，コンパイラは原則的に最内ループに対してベクトル化を行う．本事例は，ベクトル化の対象となった最内ループのベクトル長が十分でないため，多重ループを単一のループとすることでベクトル長を拡大し，高速化を行った事例である．

図 6.1.3-1 に最適化前のコードを示す．allocatable 属性が付加されており，実行時に動的に allocate される多次元配列は，演算範囲が連続であってもコンパイラがそれを判断できないため多重ループが一重化されない．この事例では，ベクトル化されるループのループ長は，配列の 1 次元目のサイズ(in)で定義され，その大きさは 441 である．そこで，配列式を多重ループに書き下し，メモリ上で連続する領域を単一のループで走査することによりベクトル長を拡大して実行時間の短縮を図る．

図 6.1.3-2 に示す最適化前の FTRACE 情報では，ベクトル演算率は 97% 以上と比較的高い値となっており，平均ベクトル長も 216.5 と良好な値となっているが，平均ベクトル長の最大値 256 に近づけることができればさらなる性能向上が期待できる．

```
! 最適化前

V======> a(1:in,1:jn,kst:ked) = (var1 + var2)/var3
```

図 6.1.3-1 ループ一重化前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
1800	0.269(0.1)	0.149	76165.7	0.3	97.35	**216.5**	0.266	0.003	0.000	0.00	【最適化前】

図 6.1.3-2 ループ一重化前の FTRACE 情報

(2) 最適化内容

図 6.1.3-3 に最適化後のコードを示す. 配列式を多重ループに書き下し, メモリ上で連続する領域を単一のループで走査することによりループの一重化を行う. 最内ループに V マークが付加され, 二重ループが一重化されたことが確認できる.

```
! 最適化後

 +------> do k=kst,ked
 |V------>   do i=1,in*jn          <-- ループ一重化
 ||            a(i,1,k) = (var1 + var2)/var3
 |V-----     enddo
 +------  enddo
```

図 6.1.3-3 ループ一重化後のコード

(3) 性能分析

図 6.1.3-4 に最適化前後の性能値を示す.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
1800	**0.269**(0.1)	0.149	76165.7	0.3	97.35	**216.5**	0.266	0.003	0.000	0.00	【最適化前】
1800	**0.262**(0.3)	0.146	77147.6	0.0	98.64	**255.9**	0.259	0.003	0.000	0.00	【最適化後】

図 6.1.3-4 ループ一重化前後の FTRACE 情報

ループが一重化されたことで, 平均ベクトル長が 216.5 から 255.9 に増加している. 元々ベクトル長は比較的大きかったため, 最適化適用による実行時間の短縮は 4% 程度であった.

6.1.4. ループの一重化による高速化(2)

(1) 最適化方針

本事例は C 言語プログラムにおけるループ一重化の事例である. 図 6.1.4-1 に最適化前のソースコードを示す. 本事例は幾何的マルチグリッド法を用いた標準ベンチマークである HPGMG-FV (https://hpgmg.org/) から抜粋したものであり, 実行時間の多くを占める Gauss-Seidel Red-Black 法の一部である. 本コードは三重ループによって構成され, コンパイラは最内のインデックス i のループをベクトル化している. しかし, 問題サイズによってはループのループ長が短く, 平均ベクトル長が不足する. そこで, インデックス i のループと j のループを一重化し, 平均ベクトル長の拡大を試みる.

図 6.1.4-2 に最適化前の FTRACE 情報を示す. 最適化前は平均ベクトル長が 18.1 と短いことがわかる.

```
! 最適化前

S---->   for(k=klo;k<khi;k++){
|          const double * __restrict__ RedBlack =
|            level->RedBlack_FP + ghosts*(1+jStride) + kStride*((k^color000)&0x1);
|+--->     for(j=jlo;j<jhi;j++){
||V-->       for(i=ilo;i<ihi;i++){
|||                int ij  = i + j*jStride;
|||                int ijk = i + j*jStride + k*kStride;
|||   F            double Ax  = apply_op_ijk(x_n);
|||   F            x_np1[ijk] = x_n[ijk] + RedBlack[ij]*Dinv[ijk]*(rhs[ijk]-Ax);
||V--         }
|+---       }
S----     }
```

ループ一重化

図 6.1.4-1 ループ一重化前のソースコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT	E.%	PROC.NAME
533950	419.342(50.8)	0.785	13120.6	7605.5	88.79	**18.1**	335.709	14.357	0.000	90.65		【最適化前】

図 6.1.4-2 ループ一重化前の FTRACE 情報

(2) 最適化内容

図 6.1.4-3 に最適化後のソースコードを示す. 本コードでは, #pragma _NEC collapse 指示行をループに適用してもコンパイラが自動的に一重化を実施しなかったため, 手動で一重化を実施し, インデックス ij のループを作成する. 編集リストより, コンパイラがインデックス ij のループをベクトル化したことがわかる.

```
! 最適化後

V---->   for(k=klo;k<khi;k++){
|          const double * __restrict__ RedBlack =
|          level->RedBlack_FP + ghosts*(1+jStride) + kStride*((k^color000)&0x1);
|V--->     for(int ij=ilo+jlo*jStride;ij<ihi+jhi*jStride;ij++){
||           if (ij % jStride < ilo || ihi <= ij % jStride) continue;
||           int ijk    = ij + k*kStride;
||    F      double Ax  = apply_op_ijk(x_n);
||    F      x_np1[ijk] = x_n[ijk] + RedBlack[ij]*Dinv[ijk]*(rhs[ijk]-Ax);
|V---       }
V----     }
```

図 6.1.4-3 ループ一重化後のソースコード

(3) 性能分析

図 6.1.4-4 に最適化前後の FTRACE 情報を示す.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT	E.%	PROC.NAME
533950	**419.342**(50.8)	0.785	13120.6	7605.5	88.79	**18.1**	335.709	14.357	0.000	90.65		【最適化前】
609138	**280.382**(38.5)	0.460	32236.0	17977.5	97.94	**147.1**	188.292	15.102	0.000	91.65		【最適化後】

図 6.1.4-4 ループ一重化後の FTRACE 情報

平均ベクトル長が 18.1 から 147.1 に拡大しており，ループ一重化によってループ長が大幅に拡大している．また，実行時間も 419.4 秒から 280.4 秒に短縮されている．

6.1.5. ループ入れ替えによる高速化（1）

(1) 最適化方針

コンパイラの自動ベクトル化機能により最内ループがベクトル化された場合でも，そのループ長が 256 に満たない場合にはベクトル演算器の性能が十分に発揮されない場合がある．本事例は多重ループのループ順を入れ替えることで平均ベクトル長を増加させ，演算時間を短縮する手法である．

図 6.1.5-1 に最適化前のコードを示す．本コードは DO ループの終値である in と jn はそれぞれ 32 および 5,840 であり，最内側のループはベクトル化されているがループ長は 256 よりも小さい．一方その外側のループ長は 5,840 で，十分に長いループ長となっているので，一時配列を導入して演算結果を DO ループのインデックスに依存しない形で保存することで，最内ループと外側ループのループ入れ換えにより実行時間の短縮を図る．

図 6.1.5-2 に示した FTRACE 情報から，最適化前は平均ベクトル長が 56.8 と低く，最内ループがベクトル化されている場合でもベクトルプロセッサの性能を十分に発揮できていないことが分かる．

```
! 最適化前

|V----->    do j=1,jn
||
||            var1=0.0d0
||
||V---->       do i=1,in
|||    G        if (a(i,j).ne.0) var1=var1+dble(b(i,j))*c(a(i,j))
||V----        enddo
||
||            if (var1.ne.0.0d0) d(J)=exp(min(var1,var2))
||
|V-----     enddo
```

jn = 5,840
in = 32

図 6.1.5-1 ループ入れ替え前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
10000	11.799(100.0)	1.180	3000.9	470.2	97.14	**56.8**	11.801	0.000	0.000	100.00	【最適化前】

図 6.1.5-2 ループ入れ替え前の FTRACE 情報

(2) 最適化内容

図 6.1.5-3 に最適化後のコードを示す．ループ入れ替え前のコードでは，変数 var1 は内側の i のループ内で足しこみ演算が行われ，外側の j のループがまわるごとに更新される．つまり，j ごとに i のループの足しこみ演算の結果を持っておく必要がある．そこで，最内 DO ループとその外側の DO ループの順序を入れ替えるために，一時配列 work1(jn) を導入した．この配列に 1 から jn

までの足し込み演算の結果を格納することで，jのループを最内側としてベクトル化が可能となる．最内側になったｊのループにＶマークが付加されており，ベクトル化されたことがわかる．

```
! 最適化後

          double precision::work1(jn)

|V=====>    work1(:)=0.0d0
|
|+----->    do i=1,in
||V---->      do j=1,jn
|||    G       if (a(i,j).ne.0) work1(j)=work1(j)+dble(b(i,j))*c(a(i,J))
||V----       enddo
|+-----     enddo
|
|V----->    do j=1,jn
||           var1 = work1(j)
||           if (var1.ne.0.0d0) d(j)=exp(min(var1,var2))
|V-----     enddo
```

図 6.1.5-3 ループ入れ替え後のコード

(3) 性能分析

図 6.1.5-4 に，最適化前後の性能値を示す．

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
10000	**11.799**(100.0)	1.180	3000.9	470.2	**97.14**	**56.8**	11.801	0.000	0.000	100.00	【最適化前】
10000	**0.448**(99.7)	0.045	56565.1	12387.7	**99.41**	**254.1**	0.450	0.000	0.000	100.00	【最適化後】

図 6.1.5-4 ループ入れ換え前後の FTRACE 情報

最内ループとその外側のループを入れ替えたことにより，ベクトル演算率が 97.14% から 99.41 %に，平均ベクトル長が 56.8 から 254.1 まで増加し，実行時間が 11.8 秒から 0.5 秒に短縮されている．

6.1.6. ループ入れ替えによる高速化(2)

(1) 最適化方針

本事例は，2 次元配列による配列式がベクトル化されるさいに，要素数の少ない次元にアクセスするループがベクトル化の対象となってされてしまう事例である．

図 6.1.6-1 に最適化前のコードを記載する．2 次元配列 a および b の計算が配列式として記述されている．配列の 1 次元目，2 次元目の要素数はそれぞれ in，jn となっており，in = 4，jn = 5,067,629 である．この配列式はコンパイラにより二重ループに展開されるが，最内ループは配列の 1 次元目にアクセスするように展開されるため，ベクトル化対象のループのループ長が 4 になってしまう．そこで，ループ長が非常に大きい 2 次元目がベクトル化の対象となるように最適化を行い平均ベクトル長を拡大する．図 6.1.6-2 に最適化前の FTRACE 情報を記載する．平均ベクトル長が 4 となっており，要素数の小さい 1 次元目でベクトル化されていることがわかる．

```
!最適化前
! 配列の allocate 部                    1 次元目の要素数が小さい（IN=4, JN=5,067,629）
            allocate(a(in, jn), b(in, jn))
! 演算部
 |V=====>F  a = a + b * var1
```

図 6.1.6-1 ループ入れ替え前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
200	17.211(99.0)	86.055	4887.8	471.1	33.73	**4.0**	17.210	0.001	0.000	0.00	【最適化前】

図 6.1.6-2 ループ入れ替え前の FTRACE 情報

(2) 最適化内容

要素数が大きい 2 次元目でベクトル化を行うためには，いくつか方法が考えられる．一つ目は，配列の次元を入れ替え，要素数が大きい次元を 1 次元目に変更する方法である．配列の次元を入れ替えたコード例を図 6.1.6-3 に記載する．演算部の配列式の記載は変わっていないが，配列を allocate するさいに，1 次元の要素数が jn，2 次元目の要素数が in になるようにしている．この配列の次元入れ替えにより，ベクトル化対象ループのループ長が拡大する．二つ目の方法は，配列式をループ構造に書き換え，要素数の小さい次元にアクセスするループをループ展開することで，ループ長の長いループでベクトル化する方法である．ループ展開を適用したコード例を図 6.1.6-4 に記載する．配列式を二重ループ構造に書き下すことで，コンパイラはループ長の短い最内側ループを自動的にループ展開し，外側ループでベクトル化を行っている．

```
!最適化後（配列の次元入れ替え）
! 配列の allocate 部                    1 次元目と 2 次元目を入れ替え（IN=4, JN=5,067,629）
            allocate(a(jn, in), b(jn, in))
! 演算部
 |V=====>F  a = a + b * var1
```

図 6.1.6-3 配列の次元入れ替え後のコード

```
!最適化後（ループ展開）
! 配列の allocate 部
            allocate(a(in, jn), b(in, jn))
! 演算部
 |V----->   do j = 1, jn                  <-- 最内側ループがループ展開され，外側ループでベクトル化
 ||*---->    do i = 1, in
 |||    F    a(i, j) = a(i, j) + b(i, j) * dt
 ||*----     enddo
 |V-----    enddo
```

図 6.1.6-4 ループ展開後のコード

(3) 性能分析

図 6.1.6-5 に, 最適化前後の性能値を示す.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
200	**17.211**(99.0)	86.055	4887.8	471.1	**33.73**	**4.0**	17.210	0.001	0.000	0.00	【最適化前】
200	**0.412**(69.7)	2.059	69612.9	19691.5	**99.01**	**256.0**	0.410	0.002	0.000	0.00	【次元入れ替え】
200	**0.282**(61.2)	1.410	72197.6	28749.8	**99.55**	**256.0**	0.282	0.001	0.000	0.06	【ループ展開】

図 6.1.6-5 最適化前後の FTRACE 情報

配列の次元入れ替えおよびループ展開により, 要素数が大きいループがベクトル化対象になり, 平均ベクトル長が 4 から 256 に向上している. さらにベクトル演算率も 33.7% から 99% 以上に改善している. 次元入れ替えの場合には, ループ長の大きいループが最内側ループ, ループ長の短いループが外側ループの二重ループの構造として処理されるが, ループ展開の場合には, 最内側のループが展開されることで一重ループの構造となりループ呼び出しのオーバヘッドが削減され, 最も性能が高くなっていると考えられる. 最適化の適用により, 実行時間は 17.3 秒から 0.5 秒および 0.3 秒とおよそ 40 倍以上短縮されている.

6.1.7. ループブロッキングによる高速化

(1) 最適化方針

本事例は, 多重ループ構造において, ベクトル化対象の内側ループのループ長が小さく, 外側ループのループ長が大きい場合に, 外側ループをブロッキングしてベクトル長を拡大する事例である.

図 6.1.7-1 に最適化前のコードを示す. 二重DOループの最内ループは, コンパイラの自動ベクトル化機能によりベクトル化されている. 図 6.1.7-2 に示した最適化前の FTRACE 情報では, 平均ベクトル長が 134.0 となっている. これは, 変数 np が 6 であるため, ベクトル化の対象となっている 2 つのループのうち, ループインデックス n2 のループ長は 216 であるが, ループインデックス m3 のループ長は 6 であるため, ループインデックス m3 のループがボトルネックとなっているためである. 一方, 最外ループのループ長 na は 11,430 と長いため, 最外ループをブロッキングして最内ループに移動することで, ベクトル長の伸長による性能改善を図る.

```
!最適化前

V======>    i = 0.d0
          !nec$ novector
+------>    do n = 1,na          <-- 最外側ループのループ長 na = 11,430
|             var2 = int(a(n)) - np
|             var3 = int(b(n)) - np
|             var4 = int(c(n)) - np
|
|C----->       do m3 = 1,np      <-- 内側ループのループ長 np = 6
||              var2 = var2 + 1
||              k = var2 + 1 + (var5 - isign(var5,var2))/2
||              if ((k < var6) .or. (var7 < k)) then
||                d(m3) = 0
||              else
||                d(m3) = k
||              end if
||              var3 = var3 + 1
||              e(m3) = var3 + 1 + (var8 - isign(var8,var3))/2
||              var4 = var4 + 1
||              f(m3) = var4 + 1 + (var9 - isign(var9,var4))/2
|C-----        end do
|V----->       do n2 = 1,np*np*np   <-- 内側ループのループ長 np * np * np = 216
||               m3=(n2-1)/(np*np)+1
||               m2=mod(((n2-1)/np),np)+1
||               m1=mod((n2-1),np)+1
||      G        l3=d(m3)
||               if(k.eq.0) cycle
||      G        l2=e(m2)
||      G        l1=f(m1)
||      G        var10 = g(n)*h(1,l1,l2,l3)
||      G        i(1,n) = i(1,n) - var9 * var10 * j(m1,n) * k(m2,n) * l(m3,n)
||      G        i(2,n) = i(2,n) - var8 * var10 * m(m1,n) * n(m2,n) * l(m3,n)
||      G        i(3,n) = i(3,n) - var5 * var10 * m(m1,n) * k(m2,n) * o(m3,n)
|V-----        end do
+------     end do
V------>    do n = 1,na
|*=====>F      p(1:3,n) = p(1:3,n) + q(1:3)*i(1:3,n)
|*=====>F      r(1:3,n) = r(1:3,n) + q(1:3)*i(1:3,n)
V------     end do
```

図 6.1.7-1 ループブロッキング前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
501	3.301(99.9)	6.589	31639.8	8269.3	98.14	**134.0**	3.301	0.000	0.000	100.00	【最適化前】

図 6.1.7-2 ループブロッキング前の FTRACE 情報

(2) 最適化内容

図 6.1.7-3 に最適化後のコードを示す. 最外ループを変数 blksz(最大ベクトル長の 256)でループブロッキングし, ブロッキングしたループを最内ループに移動することでベクトル化の対象とする. 修正後の編集リストでは, 最適化対象の 2 つのループのどちらもブロッキングにより, より長いベクトル長が得られるループがベクトル化されている.

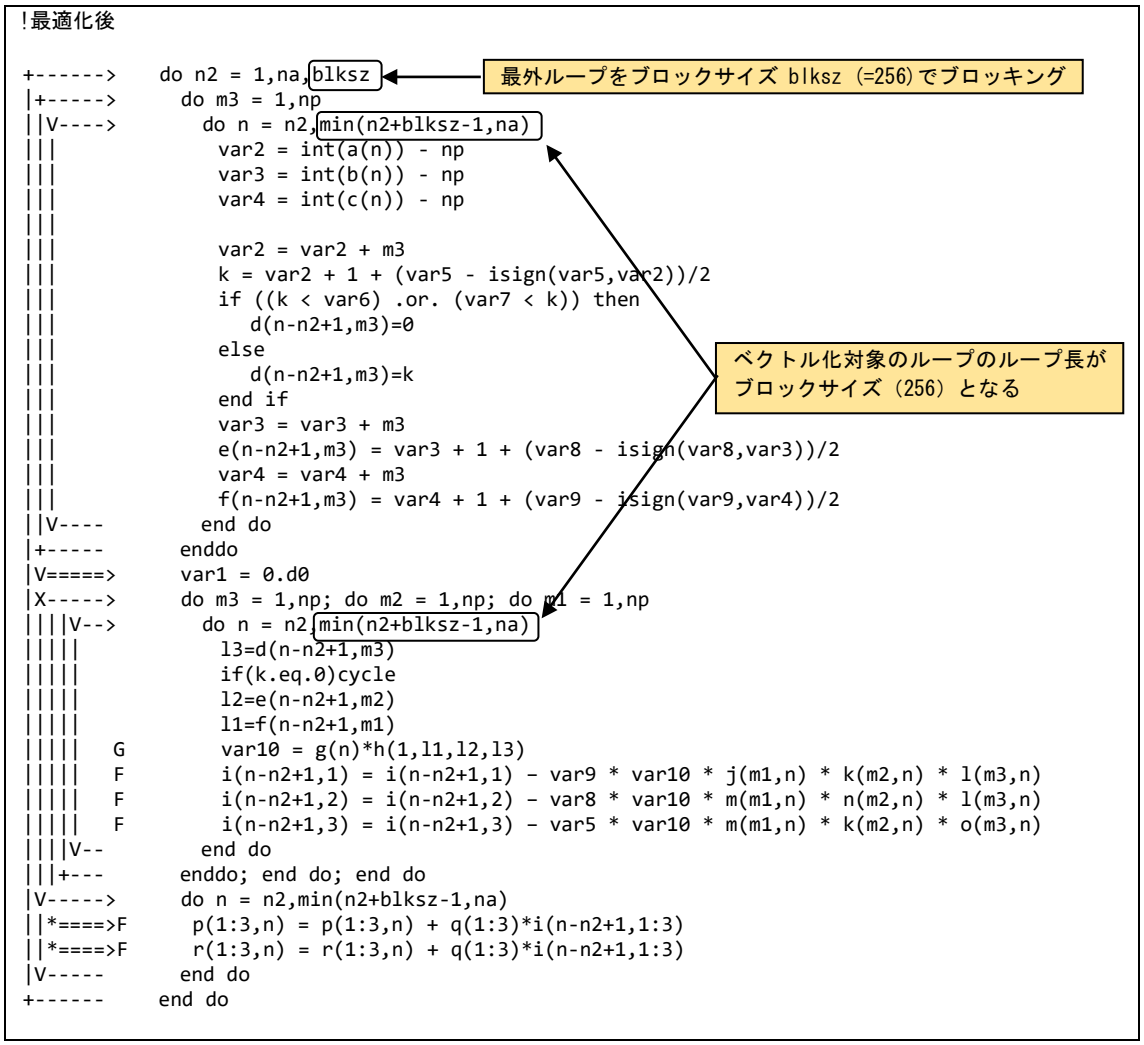

図 6.1.7-3 ループブロッキング後のコード

(3) 性能分析

図 6.1.7-4 に，最適化前後の性能値を示す.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
501	**3.301**(99.9)	6.589	31639.8	8269.3	98.14	**134.0**	3.301	0.000	0.000	100.00	【最適化前】
501	**0.500**(99.4)	0.998	108857.5	47163.6	99.63	**254.0**	0.484	0.006	0.000	100.00	【最適化後】

図 6.1.7-4 ループブロッキング前後の FTRACE 情報

ブロッキングにより分割したループ（最大ループ長が blksz）を最内ループに移動することにより，ベクトルループのループ長が伸長するため，平均ベクトル長が 134.0 から 254.0 に増加し，実行時間が 3.3 秒から 0.5 秒に短縮されている.

6.1.8. ループ分割による高速化(1)

(1) 最適化方針

最内ループ内にベクトル化が不可能な処理が含まれる場合，コンパイラはベクトル化が不可能な処理を除くその他処理について，自動ループ分割等による部分ベクトル化を試みる．しかし，ループ分割適用後もコードの意味が保持されるか否かをコンパイラが判断することは一般的に困難であり，自動ループ分割による部分ベクトル化が適用されないケースは多く発生する．本事例は，ベクトル化が不可能な処理を含むループを明示的に分割し，ベクトル化が不可な処理をループ外で行うことでベクトル化を促進し，高速化を行った例である．

図 6.1.8-1 に最適化前のコードを示す．変数 var6 の値はループ内での総和計算により算出される変数 var[1-5]の総和であるためベクトル化不可となっている．しかし，var6 の値はこの三重ループ内では参照されないため，var6 の計算をこの三重ループ外で行っても，元のコードで期待されているものと同様の結果を得ることができる．

そこで，三重ループを var6 算出処理の前後で分割することでベクトル化を促進して実行時間の短縮を図る．図 6.1.8-2 に示す最適化前の FTRACE 情報では，ベクトル演算率が 62.8% となっており十分でない．ループ分割によってベクトル化を促進することで性能向上が期待できる．

```
! 最適化前

+------> do k=kst,ked
|+------>    do j=1,jn
||+---->       do i=1,in
[…]
|||               var1 = var1 + […]
|||               var2 = var2 + […]
|||               var3 = var3 + […]
|||               var4 = var4 + […]
|||               var5 = var5 + […]
|||               var6 = var1 + var3 - var2 - var4 - var5   <-- ベクトル化不可
|||V--->           do l=1,13
||||                a(l) = a(l) &
||||                  +(var7*b(i,j,k)*c(i,j,k)*d(e(i,j,k))*(var8**3))*f(i,j,k,l)
||||                g(l) = g(l) +(var7*c(i,j,k)*d(e(i,j,k))*(var8**3))*f(i,j,k,l)
|||V---            enddo
||+----          enddo
|+-----       enddo
+------     enddo
```

図 6.1.8-1 ループ分割前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
1800	9993.424(94.3)	5551.902	1792.0	284.5	**62.83**	93.9	2958.823	5125.702	93.528	100.00	【最適化前】

図 6.1.8-2 ループ分割前の FTRACE 情報

(2) 最適化内容

図 6.1.8-3 に最適化後のコードを示す．太字で強調した DO 文および END DO 文を追加することでループを分割し，ベクトル化の阻害要因となっていた var6 の算出をループ外で行うことでループ変数 i のループがベクトル化されている．最内ループである i ループに V マークが付加され，当該ループがベクトル化されたことが確認できる．

```
! 最適化後

+------> do k=kst,ked
|+------>    do j=1,jn
||V---->       do i=1,in
[…]
|||     F         var1 = var1 + [...]
|||     F         var2 = var2 + [...]
|||               var3 = var3 + [...]
|||     F         var4 = var4 +[...]
|||     F         var5 = var5 +[...]
||V----         enddo
|+-----     enddo
+------  enddo
           var6 = var1 + var3  - var2 - var4 - va
+------> do k=istrz, iendz
|+------>    do j=1,jn
||V---->       do i=1,in
|||V--->          do l=1,13
||||               a(l) = a(l) &
||||                 +(var7*b(i,j,k)*c(i,j,k)*d(e(i,j,k))*(var8**3))*f(i,j,k,l)
||||               g(l) = g(l) +(var7*c(i,j,k)*d(e(i,j,k))*(var8**3))*f(i,j,k,l)
|||V---           enddo
||V----         enddo
|+-----     enddo
+------  enddo
```

var6 の計算はループ外で実施

図 6.1.8-3 ループ分割後のコード

(3) 性能分析

図 6.1.8-4 に, 最適化前後の性能値を示す.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
1800	**9993.424**(94.3)	5551.902	1792.0	284.5	**62.83**	93.9	2958.823	5125.702	93.528	100.00	【最適化前】
1800	**316.232**(94.6)	175.684	23140.2	15106.8	**95.73**	56.3	255.830	2.911	87.272	40.21	【最適化後】

図 6.1.8-4 ループ分割前後の FTRACE 情報

ループ分割によるベクトル化促進により, ベクトル演算率が 62.8% から 95.7% に増加し, 実行時間が 9,993.5 秒から 316.3 秒に短縮されている.

6.1.9. ループ分割による高速化(2)

(1) 最適化方針

本事ではベクトル化の対象となる DO ループ内にリストベクトル(間接参照)による配列の参照・定義があるためベクトル化が行われていない. そのため, ループを分割してベクトル化可能なループとベクトル化不可のループに分割してループの部分的なベクトル化を促進する例である.

図 6.1.9-1 に最適化前のコードを示す. ループ内で参照・定義している配列 g, h, n および k のインデックスは, スカラ変数 i と配列 a の値である jj となっている. jj が間接参照(リストベクトル)となるため, i と jj に重なりがあるかコンパイラは判断できずベクトル化の阻害要因となる. そのため, ベクトル化が可能な部分とベクトル化が不可能な部分にループ分割を行う. また, ベクトル化不可のループに対して, LIST_VECTOR 指示行を指定することで可能な限りベクトル化を行う.

図 6.1.9-2 の FTRACE 情報から，最適化前のベクトル演算率は 0.0% でありベクトル化ができていないことが分かる．

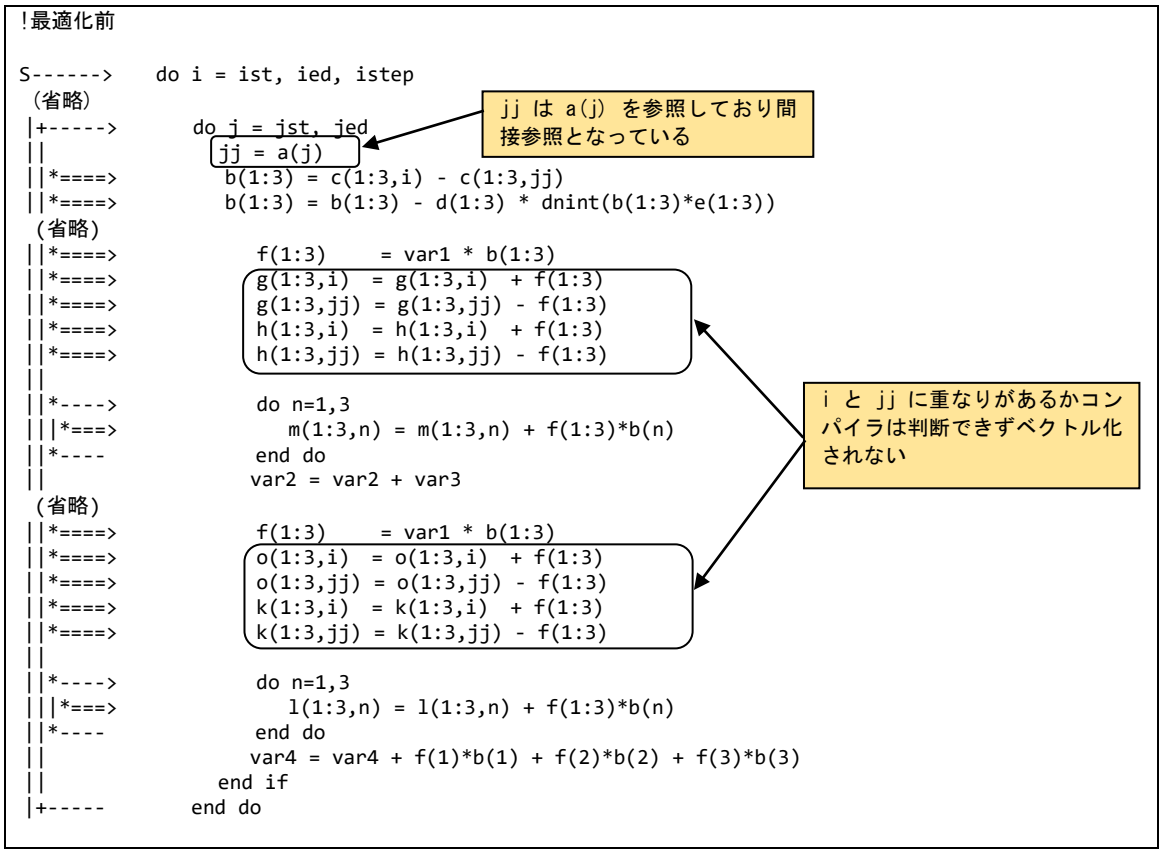

図 6.1.9-1 ループ分割前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
714927	61.087(86.5)	0.085	1683.8	377.3	**0.00**	0.0	0.000	15.439	0.000	0.00	【最適化前】

図 6.1.9-2 ループ分割前の FTRACE 情報

(2) 最適化内容

図 6.1.9-3 に最適化後のコードを示す．配列 f の値を作業配列 work1，変数 b の値を作業配列 work2 に一時保存しループ分割を行う．分割した前半のループは，配列 g，h，j および k のインデックスはスカラ変数 i のみとなるため，コンパイラは総和のマクロを適用してベクトル化を行う．分割した後半のループは，リストベクトルによりベクトル化不可となるため，LIST_VECTOR 指示行を挿入し可能な限りのベクトル化を行う．

```
!最適化後

|V=====>       work1 = 0.0d0
|V=====>       work2 = 0.0d0
|V----->       do j = jst, jed
||               jj = a(j)
||*====>G        b(1:3) = c(1:3,i) - c(1:3,jj)
||*====>F        b(1:3) = b(1:3) - d(1:3) * dnint(b(1:3)*e(1:3))
 (省略)
||*====>            f(1:3)     = var1 * b(1:3)
||*====>            work1(1:3,j,1) = f(1:3)
||                   work2(j,1)=1d0
||*====>            g(1:3,i)  = g(1:3,i)  + f(1:3)
||*====>            h(1:3,i)  = h(1:3,i)  + f(1:3)
||
||*---->            do n=1,3
|||*===>F              m(1:3,n) = m(1:3,n) + f(1:3)*b(n)
||*----              end do
||                  var2 = var2 + var3
 (省略)
||*====>            f(1:3)     = var1 * b(1:3)
||*====>            work1(1:3,j,2) = f(1:3)
||                   work2(j,2)=1d0
||*====>            o(1:3,i)  = o(1:3,i)  + f(1:3)
||*====>            k(1:3,i)  = k(1:3,i)  + f(1:3)
||
||*---->            do n=1,3
|||*===>F              l(1:3,n) = l(1:3,n) + f(1:3)*b(n)
||*----              end do
||      F           var4 = var4 + f(1)*b(1) + f(2)*b(2) + f(3)*b(3)
||                end if
|V-----        end do
|         !nec$ list_vector
|V----->       do j = jst, jed
||               jj = a(j)
||*====>C        g(1:3,jj) = g(1:3,jj) - work1(1:3,j,1)*work2(j,1)
||*====>C        h(1:3,jj) = h(1:3,jj) - work1(1:3,j,1)*work2(j,1)
||*====>C        o(1:3,jj) = o(1:3,jj) - work1(1:3,j,2)*work2(j,2)
||*====>C        k(1:3,jj) = k(1:3,jj) - work1(1:3,j,2)*work2(j,2)
|V-----       END DO
```

i に関する処理と jj に関する処理に分割

図 6.1.9-3 ループ分割後のコード

(3) 性能分析

図 6.1.9-4 に, 最適化前後の性能値を示す.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
714927	**61.087**(86.5)	0.085	1683.8	377.3	**0.00**	0.0	0.000	15.439	0.000	0.00	【最適化前】
714927	**7.373**(40.7)	0.010	27226.1	9745.3	**99.39**	191.2	7.261	0.052	0.149	99.67	【最適化後】

図 6.1.9-4 ループ分割前後の FTRACE 情報

ループ分割により, ベクトル化可能部分はベクトル化が行われ, ベクトル化不可部分は LIST_VECTOR 指示行により可能な限りのベクトル化が行われた. 最適化の効果で, ベクトル演算率は 99.4% に向上し, 実行時間が 61.1 秒から 7.4 秒に短縮されている.

6.1.10. 依存関係解消による高速化(1)

(1) 最適化方針

コンパイラはプログラムを解析してベクトル命令で実行可能な部分を自動的に検出し，その部分に対してベクトル命令を生成する．対象となるループ，または，配列式にベクトル化可能な部分と不可の部分が混在している場合は，ベクトル化可能な部分のみをベクトル化する．このベクトル化法は部分ベクトル化と呼ばれる．本事例は部分ベクトル化された DO ループに対し指示行 IVDEP を指定することで，ベクトル化不可の依存関係がないと仮定してコンパイラにベクトル化する事を許可する．依存関係があるループをベクトル化すると結果が不正となったり，ベクトル化前後で計算結果が異なったりすることがあるので結果の確認が必要である．

図 6.1.10-1 に最適化前のコードを示す．本コードは最内 DO ループ内の演算はインデックスが配列の値で与えられる(間接参照)ためコンパイラによる依存関係の有無が判別不可能であり，「部分ベクトル化」を示す ”S” となっている．本コードでは配列 a(j) の値に重なりがないこと(単一のベクトル命令で処理される範囲で同一の値が複数回現れないこと)をユーザ側がわかっていたため，この DO ループの直前に指示行 IVDEP を指定することで依存関係が無いとし，DO ループをベクトル化することにより実行時間の短縮を図る．

図 6.1.10-2 に示した FTRACE 情報から，最適化前はベクトル演算率が 3.96% と低く，ベクトルプロセッサの性能を十分に発揮できていないことが分かる．

```
! 最適化前

|S----->    do j=1,jn1
||           i=a(j)
||           var1=b(j)*var2
||           b(i)=b(i)*exp(var1)
|S-----     enddo
|
|
|S----->    do j=1,jn2
||           i=c(j)
||           var3=d(j)*var4
||           d(i)=d(i)*exp(var3)
|S-----     enddo
```


図 6.1.10-1 指示行挿入前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
10000	5.075(100.0)	0.508	1493.8	458.1	**3.96**	250.0	0.004	0.095	0.000	100.00	【最適化前】

図 6.1.10-2 指示行挿入前の FTRACE 情報

(2) 最適化内容

図 6.1.10-3 に最適化後のコードを示す．

「部分ベクトル化」となっている 2 箇所の最内ループの直前に指示行 IVDEP を指定することで依存関係が無いとしたため，コンパイラにより DO ループがベクトル化されたことが確認できる．

なお，指数関数の演算文に表記された ”C” は，ベクトル拡散(スキャッター)命令によるベクト

ル化が行われた事を示す.

```
! 最適化後

|        !NEC$ IVDEP          <-- 指示行の挿入
|V----->    do j=1,jn1
||            i=a(j)
||            var1=b(j)*var2
||      C     b(i)=b(i)*exp(var1)
|V-----     enddo
|
|        !NEC$ IVDEP          <-- 指示行の挿入
|V----->    do j=1,jn2
||            i=c(j)
||            var3=d(j)*var4
||      C     d(i)=d(i)*exp(var3)
|V-----     enddo
```

図 6.1.10-3 指示行挿入後のコード

(3) 性能分析

図 6.1.10-4 に, 最適化前後の性能値を示す.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
10000	**5.075**(100.0)	0.508	1493.8	458.1	**3.96**	250.0	0.004	0.095	0.000	100.00	【最適化前】
10000	**0.041**(97.7)	0.004	86422.9	58458.9	**99.35**	250.0	0.041	0.000	0.000	100.00	【最適化後】

図 6.1.10-4 指示行挿入前後の FTRACE 情報

指示行 IVDEP により最内ループをベクトル化したことにより, ベクトル演算率が 3.96% から 99.35% に増加し, 実行時間が 5.1 秒から 0.041 秒に短縮されている.

6.1.11. 依存関係解消による高速化(2)

(1) 最適化方針

コンパイラは原則的に最内ループに対してベクトル化を行うが, 当該ループにベクトル化を不可能とする依存関係がないことが保証できない場合にはベクトル化を適用しない. 本事例は, ベクトル化を不可能とする依存関係がないことが明確となるようにコードを修正してベクトル化を促進し, 高速化を行った例である.

図 6.1.11-1 に最適化前のコードを示す. 配列 c の i2 番目の要素への加算が行われているが, インデックス i2 はループ変数 i1 から算出されており, コンパイラがベクトル化を不可能とする依存関係がないこと(単一のベクトル命令で処理される範囲で i2 に重なりがない, つまり, 同一の値が複数回現れないこと)を保証できない. そのためベクトル化が行われず, 図 6.1.11-2 に示した最適化前の FTRACE 情報では, ベクトル演算率が 0 % となっている. ベクトル化を不可能とする依存関係がないことが明確となるようにコードを修正し, ベクトル化を促進することによって実行時間の短縮を図る.

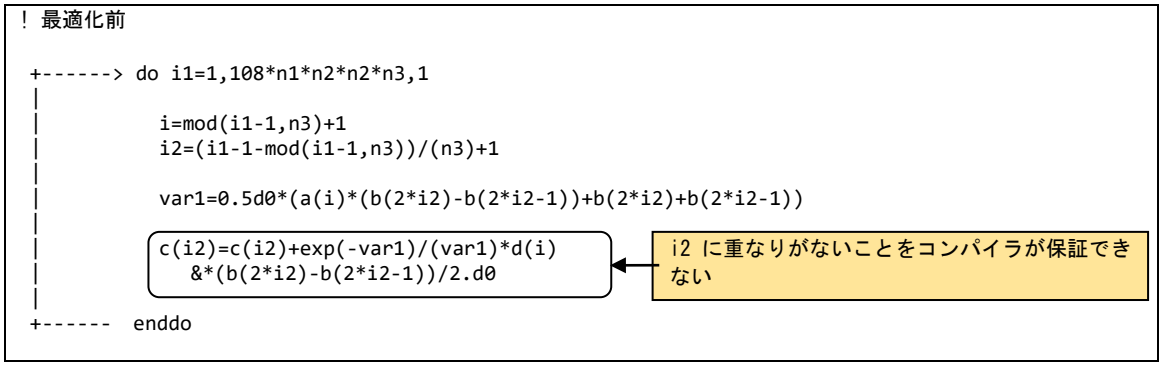

図 6.1.11-1 依存関係解消前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT	E.%	PROC.NAME
1	0.214(99.8)	214.240	1787.1	497.2	**0.00**	0.0	0.000	0.001	0.000	0.00		【最適化前】

図 6.1.11-2 依存関係解消前の FTRACE 情報

(2) 最適化内容

図 6.1.11-3 に最適化後のコードを示す. 単一のループよる処理を二重ループに書き換え, ループ変数の変換を不要とした. ベクトル化を不可能とする依存関係がないことが明確となりベクトル化が行われた. 編集リストでは最内ループがベクトル化されたことを示す V マークが付加されている.

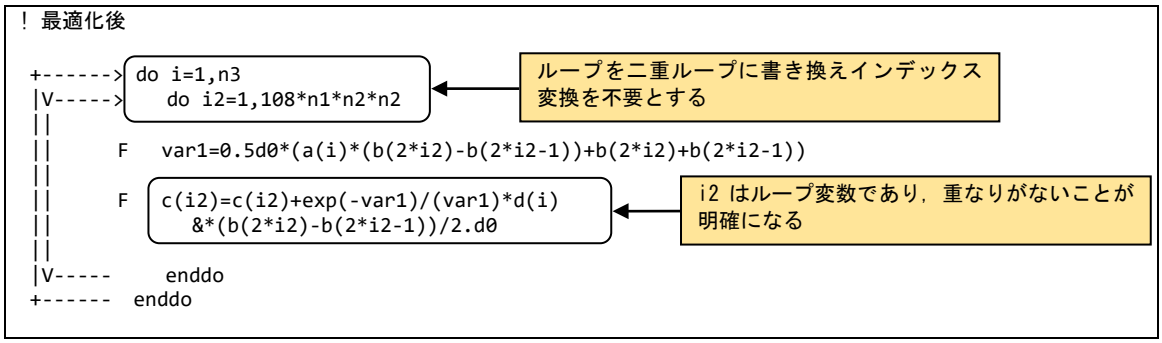

図 6.1.11-3 依存関係解消後のコード

(3) 性能分析

図 6.1.11-4 に最適化前後の性能値を示す.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT	E.%	PROC.NAME
1	**0.214**(99.8)	214.240	1787.1	497.2	**0.00**	0.0	0.000	0.001	0.000	0.00		【最適化前】
1	**0.001**(75.4)	1.034	189640.7	130537.7	**99.67**	255.9	0.001	0.000	0.000	99.99		【最適化後】

図 6.1.11-4 依存関係解消前後の FTRACE 情報

内側の DO ループがベクトル化されたことで，ベクトル演算率が 0% から 99.6% に向上し，実行時間が 0.214 秒から 0.001 秒に短縮されている．

6.1.12. インライン展開による高速化

(1) 最適化方針

一般に手続き(サブルーチンおよび関数)の呼出し回数が多い場合，その呼出し処理のためのオーバヘッドが顕著になり実行時間が長くなることがある．本事例は自動インライ展開を適用するコンパイルオプション -finline-functions のみでは，他ソースコードファイルに記述されたサブルーチンがインライン展開の対象とならない場合に，クロスファイルインライニングを指定するオプション -finline-file でソースファイル名を指定することで呼出回数の多いサブルーチンのインライン展開の対象とする手法である．

なお，インライン展開機能を利用する際に多数の手続きをインライン展開するとプログラムのコードサイズが肥大化し，命令キャッシュからコードが溢れることでプログラム全体の実行性能が低下する事がある．また，インライン展開を大きなソースコードや多数のファイルを対象とする場合，それ以降の処理量が増えコンパイル時間が延びたり，コンパイル時に使用するメモリ量が増加したりするので併せて注意が必要である．

インライン展開を行う場合は図 6.1.12-1 に示したコンパイルオプションを指定する．

また，インライン展開された手続きを確認する場合は，図 6.1.12-2 に示した最適化情報リストを出力するコンパイルオプションを指定する．

```
-finline-functions
```

図 6.1.12-1 インライン展開を行うコンパイルオプション

```
-report-inline
```

図 6.1.12-2 インライン展開モジュールの最適化情報リストを出力するコンパイルオプション

図 6.1.12-3 に示した FTRACE 情報(演算コスト上位 15 件)から，-finline-functions オプションのみを指定してインライン展開を行った場合は，○で示した呼出回数の多いサブルーチン 4 つは，インライン展開の対象外であったことが分かる．

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME	
3000	58.316(26.3)	19.439	23491.3	12649.4	98.08	112.1	55.474	0.931	6.196	22.91	SUB_P1	
3000	40.523(18.3)	13.508	73206.6	54417.5	98.88	108.2	40.510	0.010	8.542	71.49	SUB_J	
1188000	26.961(12.2)	0.023	22753.0	11741.9	98.18	105.4	23.603	0.262	10.749	99.82	SUB_C1	○
1800000	18.479(8.3)	0.010	40964.5	21112.5	98.02	107.6	18.268	0.209	14.335	99.74	SUB_C2	○
3000	17.293(7.8)	5.764	35805.3	16680.8	97.73	110.3	16.847	0.432	2.402	22.85	SUB_P2	
1800000	14.137(6.4)	0.008	54822.2	35975.3	98.46	107.7	13.611	0.526	10.485	99.00	SUB_C3	○
891000	10.869(4.9)	0.012	51806.0	28336.4	98.75	101.6	10.451	0.306	0.363	96.62	SUB_C4	○
3000	8.904(4.0)	2.968	33984.1	17567.4	98.59	108.5	8.880	0.019	0.000	3.18	SUB_K	
297000	8.072(3.6)	0.027	61181.0	36872.5	98.97	96.0	7.944	0.090	0.181	99.90	SUB_P3	
3000	6.887(3.1)	2.296	72586.1	53142.4	99.26	115.0	6.886	0.001	0.000	2.35	SUB_L	
1000	4.789(2.2)	4.789	27022.4	6859.4	96.86	112.5	4.767	0.022	0.800	36.26	SUB_F	
3001	3.314(1.5)	1.104	68264.7	12045.2	99.89	254.5	3.199	0.079	0.010	46.96	SUB_I	
1000	2.024(0.9)	2.024	37783.1	12170.5	98.17	115.0	2.023	0.001	0.000	3.31	SUB_U	
1000	0.656(0.3)	0.656	18543.1	0.0	96.38	115.0	0.656	0.000	0.000	0.91	SUB_Q	
1	0.152(0.1)	151.574	758.7	1.5	1.35	4.8	0.016	0.041	0.000	98.43	MAIN	
（省略）												
6011763	221.222(100.0)	0.037	42573.8	25672.2	98.53	109.7	212.837	3.092	54.104	63.22	total	

図 6.1.12-3 -finline-functions オプションのみ指定した場合の FTRACE 情報

(2) 最適化内容

図 6.1.12-4 に, クロスファイルインライニングオプションを指定したコンパイル方法を示す. -finline-file オプションの後に対象とするソースコードのファイル名を指定する. 複数のファイル名を指定する場合は, ファイル名を ”:(コロン)” で区切って記述する.

```
-finline-functions -report-inline -finline-file=test1.f90:test2.f90
```

図 6.1.12-4 クロスファイルインライニングオプションを指定したコンパイルオプション

(3) 性能分析

図 6.1.12-5 にクロスファイルインライニングオプションを指定した場合の FTRACE 情報(演算コスト上位 11 件)を示す.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME	
3000	68.012(33.5)	22.671	42658.7	24060.3	98.18	109.7	67.725	0.234	31.021	59.85	SUB_P1	○
3000	52.742(26.0)	17.581	31080.2	15788.0	98.25	105.6	49.890	0.298	10.773	66.89	SUB_P2	○
3000	40.982(20.2)	13.661	72385.8	53807.3	98.88	108.2	40.968	0.011	8.542	71.47	SUB_J	
297000	14.482(7.1)	0.049	44857.4	26096.8	98.77	98.0	13.684	0.138	2.921	99.85	SUB_P3	○
3000	8.838(4.3)	2.946	34235.7	17697.5	98.59	108.5	8.811	0.023	0.000	2.94	SUB_K	
3000	6.870(3.4)	2.290	72769.9	53277.0	99.26	115.0	6.868	0.002	0.000	2.14	SUB_L	
1000	5.015(2.5)	5.015	25804.5	6550.1	96.86	112.5	4.963	0.066	0.800	36.44	SUB_F	
3001	3.238(1.6)	1.079	69866.2	12328.4	99.89	255.1	3.124	0.079	0.010	47.06	SUB_I	
1000	1.992(1.0)	1.992	38385.6	12364.5	98.17	115.0	1.991	0.001	0.000	3.26	SUB_U	
1000	0.654(0.3)	0.654	18600.7	0.0	96.38	115.0	0.654	0.000	0.000	0.93	SUB_Q	
1	0.136(0.1)	136.122	799.4	1.6	1.11	5.1	0.012	0.044	0.000	98.08	MAIN	
（省略）												
330224	203.207(100.0)	0.615	46342.0	27948.2	98.55	109.7	198.887	0.917	54.078	63.37	total	

図 6.1.12-5 クロスファイルインライニングオプションを指定後の FTRACE 情報

クロスファイルインライニングオプションを指定したことにより，別ファイルに記述されたサブルーチンもインライン展開の対象となり，手続き呼出し回数の合計が 6,011,763 回から 330,224 回に減少した．これにより実行時間の合計が 221.222 秒から 203.207 秒に短縮されている．

クロスファイルインライニングオプションの指定による，呼び出し側サブルーチン（図 6.1.12-5 で○がついたサブルーチン）とインライン展開されたサブルーチンの関係は以下の通りである．

- SUB_P1 から SUB_C2，SUB_C3 の呼出し
- SUB_P2 から SUB_C1．SUB_C4 の呼出し
- SUB_P3 から SUB_C1 の呼出し

インライン展開後の呼び出し側サブルーチンの実行時間が，インライン展開前の各サブルーチンの実行時間の合計よりも短くなり，演算時間が短縮したことが確認できる．

6.1.13. ライブラリ利用による高速化（1）

(1) 最適化方針

AOBA-A では，SX-Aurora TSUBASA での計算に最適化された数学ライブラリである NEC Numeric Library（NLC）が利用可能である．この数学ライブラリをうまく活用することで，高い性能を引き出すことができる．図 6.1.13-1 に最適化前のコードを示す．ベクトル化対象のループ内には 1 要素の疑似乱数値を取得する関数 make_random（ユーザ定義関数）の呼び出しがあるため，ベクトル化が行われない．疑似乱数値を取得する処理をループ分割により抜き出し，該当の疑似乱数値の生成には NLC の乱数生成ライブラリを使用することでベクトル性能を向上させる．

図 6.1.13-2 に示す最適化前の FTRACE 情報では，ベクトル演算率は 47.6% であり十分なベクトル性能を出すことができていない．

```
!最適化前

|+------>        do j = jst, jed
||
||                jj = a(j)
||                var1 = b(jj)
||
||                var3 = c(1,i) - c(1,jj)
||                var4 = c(2,i) - c(2,jj)
||                var5 = c(3,i) - c(3,jj)
||                var3 = var3 - d(1) * anint(var3*e(1))
||                var4 = var4 - d(2) * anint(var4*e(2))
||                var5 = var5 - d(3) * anint(var5*e(3))
||
||                var6 = sqrt(var3**2 + var4**2 + var5**2)
||                f(1) = var3 / var6
||                f(2) = var4 / var6
||                f(3) = var5 / var6
||
||                var7 = g(1,i) - g(1,jj)
||                var8 = g(2,i) - g(2,jj)
||                var9 = g(3,i) - g(3,jj)
||                var10=f(1)*var7+f(2)*var8+f(3)*var9
||
||      I         var11 = make_random()
||                var12 = sqrt(-2.0d0*log(var11)*1.0d0)
||      I         var11 = make_random ()
||                var12 = var12 * cos(2.0d0*pi*var11)
||
||                var13 = (h(1,var2,var1) * f(1)) &
||                   + (h(2,var2,var1) * f(1) * var10) &
||                   + (h(3,var2,var1) * f(1) * var12)
||                var14 = (h(1,var2,var1) * f(2)) &
||                   + (h(2,var2,var1) * f(2) * var10) &
||                   + (h(3,var2,var1) * f(2) * var12)
||                var15 = (h(1,var2,var1) * f(3)) &
||                   + (h(2,var2,var1) * f(3) * var10) &
||                   + (h(3,var2,var1) * f(3) * var12)
||
||                i(1,i) = i(1,i) + var13
||                i(2,i) = i(2,i) + var14
||                i(3,i) = i(3,i) + var15
||                i(1,jj) = i(1,jj) - var13
||                i(2,jj) = i(2,jj) - var14
||                i(3,jj) = i(3,jj) - var15
||
|+-----          enddo
```

疑似乱数値を取得する関数の呼び出し

図 6.1.13-1 ライブラリ(乱数)利用前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
154207	8.492(99.6)	0.055	1275.4	162.9	**47.61**	127.0	3.594	2.125	0.000	100.00	【最適化前】

図 6.1.13-2 ライブラリ(乱数)利用前の FTRACE 情報

(2) 最適化内容

図 6.1.13-3 に最適化後のコードを示す. 疑似乱数値の生成処理をループ分割により抜き出し, 乱数値を一時配列 work1 に格納し, 分割した演算処理のループ内では work1 の値を参照するように修正する. また, 分割した演算処理のループにおいては, ループ内から関数呼び出しが削除されることでベクトル化の阻害要因が無くなるため, ベクトル化が行われる. 更に, 疑似乱数値の生成処理は NLC の乱数生成ライブラリを使用する.

```
!最適化後

|                call asl_random_generate_d(rng, 2*(j2-j1+1), work1)
|V----->          do j = jst, jed
||
||                   jj = a(j)
||      G            var1 = b(jj)
||
||      G            var3 = c(1,i) - c(1,jj)
||      G            var4 = c(2,i) - c(2,jj)
||      G            var5 = c(3,i) - c(3,jj)
||      F            var3 = var3 - d(1) * anint(var3*e(1))
||      F            var4 = var4 - d(2) * anint(var4*e(2))
||      F            var5 = var5 - d(3) * anint(var5*e(3))
||
||      F            var6 = sqrt(var3**2 + var4**2 + var5**2)
||                   f(1) = var3 / var6
||                   f(2) = var4 / var6
||                   f(3) = var5 / var6
||
||      G            var7 = g(1,i) - g(1,jj)
||      G            var8 = g(2,i) - g(2,jj)
||      G            var9 = g(3,i) - g(3,jj)
||      F            var10=f(1)*var7+f(2)*var8+f(3)*var9
||
||                   var12 = sqrt(-2.0d0*log(work1(1,j-j1+1))*1.0d0)
||                   var12 = var12 * cos(2.0d0*pi*work1(2,j-j1+1))
||
||      G            var13 = (h(1,var2,var1) * f(1)) &
||                      + (h(2,var2,var1) * f(1) * var10) &
||                      + (h(3,var2,var1) * f(1) * var12)
||      F            var14 = (h(1,var2,var1) * f(2)) &
||                      + (h(2,var2,var1) * f(2) * var10) &
||                      + (h(3,var2,var1) * f(2) * var12)
||      F            var15 = (h(1,var2,var1) * f(3)) &
||                      + (h(2,var2,var1) * f(3) * var10) &
||                      + (h(3,var2,var1) * f(3) * var12)
||
||                   i(1,i) = i(1,i) + var13
||                   i(2,i) = i(2,i) + var14
||                   i(3,i) = i(3,i) + var15
||                   i(1,jj) = i(1,jj) - var13
||                   i(2,jj) = i(2,jj) - var14
||                   i(3,jj) = i(3,jj) - var15
||
|V-----          enddo
```


図 6.1.13-3 ライブラリ(乱数)利用後のコード

(3) 性能分析

図 6.1.13-4 に最適化前後の性能値を示す．

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
154207	**8.492**(99.6)	0.055	1275.4	162.9	**47.61**	127.0	3.594	2.125	0.000	100.00	【最適化前】
154207	**0.701**(93.1)	0.005	5140.4	2593.4	**87.63**	72.5	0.120	0.377	0.004	99.81	【最適化後】

図 6.1.13-4 ライブラリ(乱数)利用前後の FTRACE 情報

ループ分割および NLC の乱数生成ライブラリを用いることによりベクトル演算率が 47.6% か 87.6% に向上し，実行時間が 8.5 秒から 0.7 秒に短縮されている．

6.1.14. ライブラリ利用による高速化(2)

(1) 最適化方針

本事例は, FFT 処理を高速化した事例である. 図 6.1.14-1 に最適化前のコードを示す. 該当処理は三次元配列に対する FFT 処理を実行しているが, FFTW の一次元 FFT を用いて実装されている. 該当 FFT 処理について, 一次元 FFT 処理から一次元多重 FFT 処理に変更することによりベクトル性能を向上させる.

図 6.1.14-2 にライブラリ(FFTW)利用前の FTRACE 情報を示す. 最適化前の FTRACE 情報では, ベクトル演算率は 51.7% であり十分なベクトル性能を出すことができていない. さらに, 平均ベクトル長が 8.2 と非常に小さな値となっていることがわかる.

```
!最適化前

! FFTW プラン生成
           call DFFTW_PLAN_DFT_1D( planB(1), in, a, a, &
               &                 FFTW_BACKWARD, FFTW_MEASURE )

! FFTW 実行
+------>    do k = 1, kn
|+----->       do j = 1, jn
||V---->          do i = 1, in
|||                 a(i) = CMPLX(b(1,i,j,k),   &
|||                      &       b(2,i,j,k),kind=8)
||V----            enddo
||                 call DFFTW_EXECUTE(planB(1))   <-- 一次元 FFT による実装
||V---->           do i = 1, in
|||                 b(1,i,j,k) = dble(a(i))
|||                 b(2,i,j,k) = aimag(a(i))
||V----            enddo
|+-----         enddo
+------      enddo
```

図 6.1.14-1 一次元多重 FFT 利用前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
500	8.144(46.6)	16.288	1888.7	563.6	**51.71**	**8.2**	3.041	1.221	0.032	100.00	【最適化前】

図 6.1.14-2 一次元多重 FFT 利用前の FTRACE 情報

(2) 最適化内容

図 6.1.14-3 に最適化後のコードを示す. 一次元 FFT 処理から一次元多重 FFT 処理に変更するため, FFTW のプラン生成処理を変更する. また, 一次元多重 FFT 処理においては, FFT 処理対象となる配列を一次元配列から多重度を含めた二次元配列に変更する必要があるため, FFT 処理を実行する際の一時配列を二次元配列に修正する. 一次元多重 FFT 処理では, 多重度方向がベクトル化の対象となる. この事例では, プラン生成時の第四引数である jn が多重度の大きさとなり, その大きさは 50 であった. 修正前の平均ベクトル長が 8 程度であったため, 平均ベクトル長の向上も期待できる.

```
! 最適化後

FFTW プラン生成
          call DFFTW_PLAN_MANY_DFT(planB_many(1), 1, in, jn, &
              &                    work1, maxn, 1, maxn, &
              &                    work1, maxn, 1, maxn, &
              &                    FFTW_BACKWARD, FFTW_MEASURE)

FFTW 実行
+------>    do k=1,kn
|U----->       do j=1,jn
||V---->          do i=1,in
|||                 work1(i,j) = CMPLX(b(1,i,j,k),   &
|||                      &             b(2,i,j,k),kind=8)
||V----           enddo
|U-----         enddo
|              call DFFTW_EXECUTE_DFT(planB_many(1), work1, work1)
|U----->       do j=1,jn
||V---->          do i=1,in
|||                 b(1,i,j,k) = dble(work1(i,j))
|||                 b(2,i,j,k) = aimag(work1(i,j))
||V----           enddo
|U-----         enddo
+------      enddo
```

一次元多重 FFT に変更するためFFTWのプラン生成処理を変更

一次元多重FFT に変更

図 6.1.14-3 一次元多重 FFT 利用後のコード

(3) 性能分析

図 6.1.14-4 に, 最適化前後の性能値を示す.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
500	**8.144**(46.6)	16.288	1888.7	563.6	**51.71**	**8.2**	3.041	1.221	0.032	100.00	【最適化前】
500	**0.377**(38.5)	0.754	20691.7	11890.8	**95.92**	**49.9**	0.274	0.026	0.095	100.00	【最適化後】

図 6.1.14-4 一次元多重 FFT 利用前後の FTRACE 情報

一次元 FFT の処理から一次元多重 FFT の処理に変更することにより, ベクトル演算率は 51.7 %から 95.9% に, 平均ベクトル長が 8.2 から 49.9 に向上し, 実行時間が 8.2 秒から 0.4 秒に短縮されている.

6.2. メモリアクセス性能向上の事例

6.2.1. メモリアクセスパターンの変更による高速化

(1) 最適化方針

間接参照によるメモリアクセスはメモリ負荷が高いことが知られている. 本事例では, メモリアクセスパターンを間接アクセスから連続アクセスに変更することでメモリ負荷を低減した最適化事例を示す.

図 6.2.1-1 に最適化前のコードを示す. 本事例で使用しているプログラムは, 3 次元空間内の特定のセルについて計算を行う. ループ長 n は計算対象のセル数を示しており, 配列 a には, 該当するセルの X, Y, Z 方向の位置情報が格納されている. そのため, まずは, それぞれのセルの位置情報を変数 i, j, k に代入し, そのあとで該当するセルの配列 c にアクセスする. つまり, 配

列 c の添え字 i, j, k がさらに配列 a を参照しており，間接参照の形になっている．配列 c にアクセスするためには，最初に添え字 i, j, k を求めるために別の配列 a にアクセスする必要があり，その結果メモリ負荷が高くなる．

アプリケーションのメモリ負荷の大きさを示す指標として B/F がある．B/F とは，1 つの浮動小数点演算を処理するために必要なメモリアクセス量を示し，この値が大きいほど 1 回の演算処理のために大量のデータをメモリから転送する必要があるといえる．しかし，必要なデータがキャッシュ上に存在する場合にはメモリにアクセスする必要はなく，キャッシュからデータを転送すればよいことになる．そこで，プログラム実行時のメモリ負荷の大きさを示す指標として Actual B/F がある．Actual B/F の場合，キャッシュヒットしたデータはカウントせず，実際にメモリまでアクセスしたデータのみをカウントして B/F を求める．Actual B/F は FTRACE の出力結果のうち，REQ. ST B/F と ACT. VLD B/F の和として計算できる．図 6.2.1-2 の最適化前の FTRACE 情報から，最適化前の Actual B/F は 1.6 であることがわかる．AOBA-A のシステム B/F は 0.62 であり，Actual B/F がシステム B/F 以上である場合，そのプログラムのメモリ負荷は高いといえる．

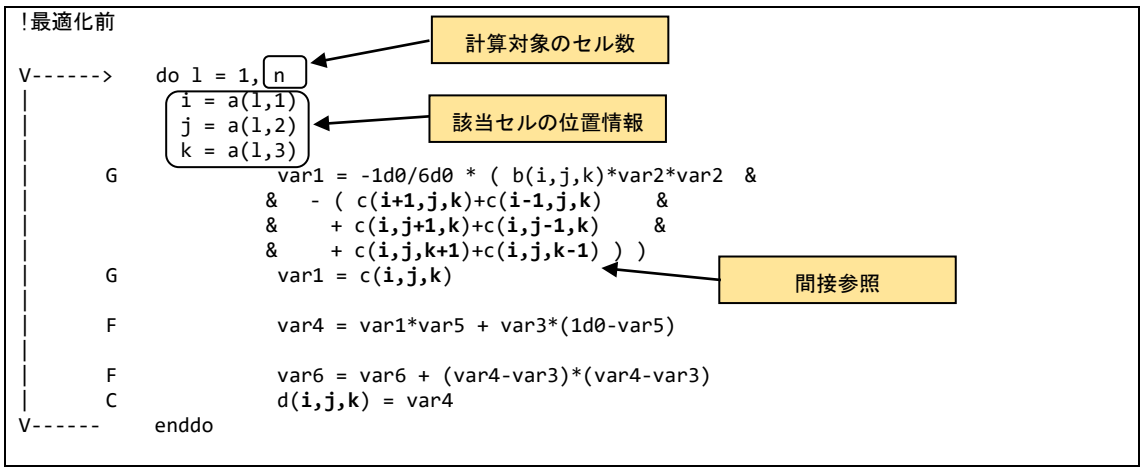

図 6.2.1-1 メモリアクセスパターン変更前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR .. TIME ..	REQ. B/F	REQ.ST B/F	REQ.LD B/F	ACT.VLD B/F	FLOP COUNT	FMA ELEM.	PROC.NAME
408000	627.070	1.537	29956.1	5443.0	99.93	256.0	626.687	3.00	**0.25**	2.75	**1.35**	3413144816976	639944916183	【最適化前】

図 6.2.1-2 メモリアクセスパターン変更前の FTRACE 情報

(2) 最適化内容

図 6.2.1-3 に最適化後のコードを示す．ist, ied は X 方向の全領域を示しており，同様に jst, jed, kst, ked は Y, Z 方向の全領域を示す．つまり，最適化後のコードは，計算対象のセルだけにアクセスするのではなく，3 次元空間内の全セルにアクセスしている．計算対象のセルについては事前に配列 work1 の値に，0 より大きい値を設定しておくことで，計算時に配列 work1 の値で計算対象のセルかどうかを判断している．最適化後のコードではメモリアクセスのパターンが連続アクセスとなるため，メモリ負荷を低減することができる．

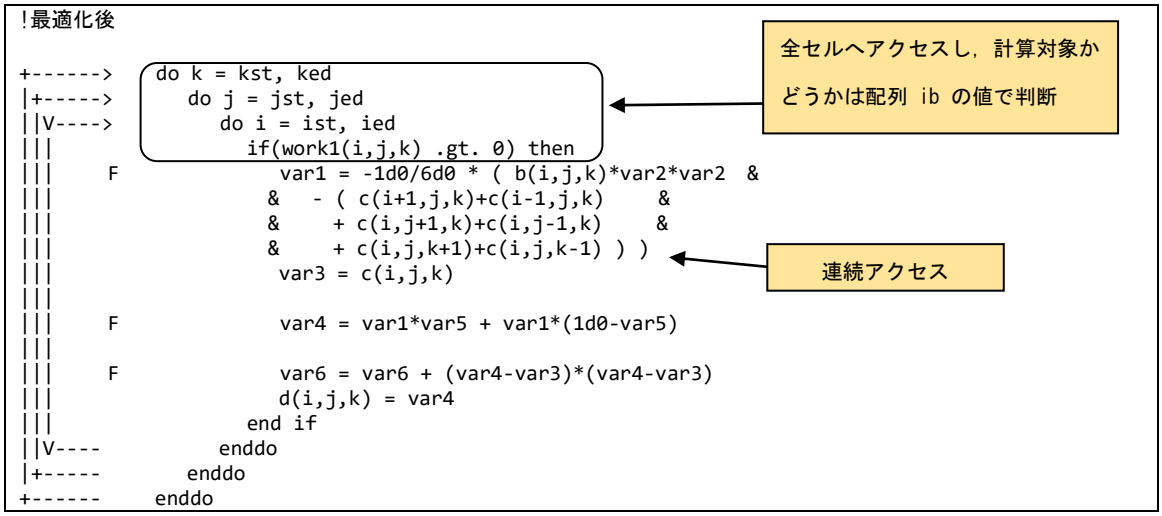

図 6.2.1-3 メモリアクセスパターン変更後のコード

(3) 性能分析

図 6.2.1-4 に最適化前後の性能値を示す.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR .. TIME ..	REQ. B/F	REQ.ST B/F	REQ.LD B/F	ACT.VLD B/F	FLOP COUNT	FMA ELEM.	PROC.NAME
408000	**627.070**	1.537	29956.1	5443.0	99.93	256.0	626.687	3.00	**0.25**	2.75	**1.35**	3413144816976	639944916183	【最適化前】
408000	**166.666**	0.408	45468.3	21838.9	96.02	62.0	165.700	2.35	**0.24**	2.12	**0.83**	3639804312000	641728512000	【最適化後】

図 6.2.1-4 メモリアクセスパターン変更前後の FTRACE 情報

メモリアクセスパターンを変更することで, Actual B/F が 1.6 から 1.07 に減少し, 実行時間が 627.1 秒から 166.7 秒に短縮されている.

6.2.2. メモリアクセス効率化による高速化

(1) 最適化方針

多重ループのループ順を入れ替えることで, 性能が大きく向上する場合がある. 例えば, 最内側ループ(ベクトル化対象ループ)のループ長が外側ループのループ長に比べて極端に小さい場合, ループを入れ替えることで, ベクトル長が長くなり性能の改善が期待できる. このように, 多重ループのそれぞれのループのループ長に極端な違いがある場合には, どのループをベクトル化の対象とすれば性能を引き出せるか容易に検討することができる. 一方で, ループ長に大きな違いがない場合には, 配列へのアクセスパターンの違いにより性能が変化すると考えられる. 本事例では, 多重ループのループ長がすべて同じだった場合に, 最適なループ順番を検討した事例である.

図 6.2.2-1 に最適化前のコードを示す. 3 つの三重ループ(ループ 1, ループ 2, ループ 3)は, ループ長がすべて 128 であるが, 配列のアクセスのパターンが異なっている. 例えば, ループ 1 の場合, ベクトル化されたループのループ変数 i は, ロードする配列 c について 1 次元目にアクセスするため, 連続アクセスとなる. 一方で, ストアする配列 a に関しては 3 次元目にアクセスするため

ストライドアクセスとなる. ループ 2, ループ 3 に関してはロードする配列がストライドアクセス, ストアする配列が連続アクセスとなる.

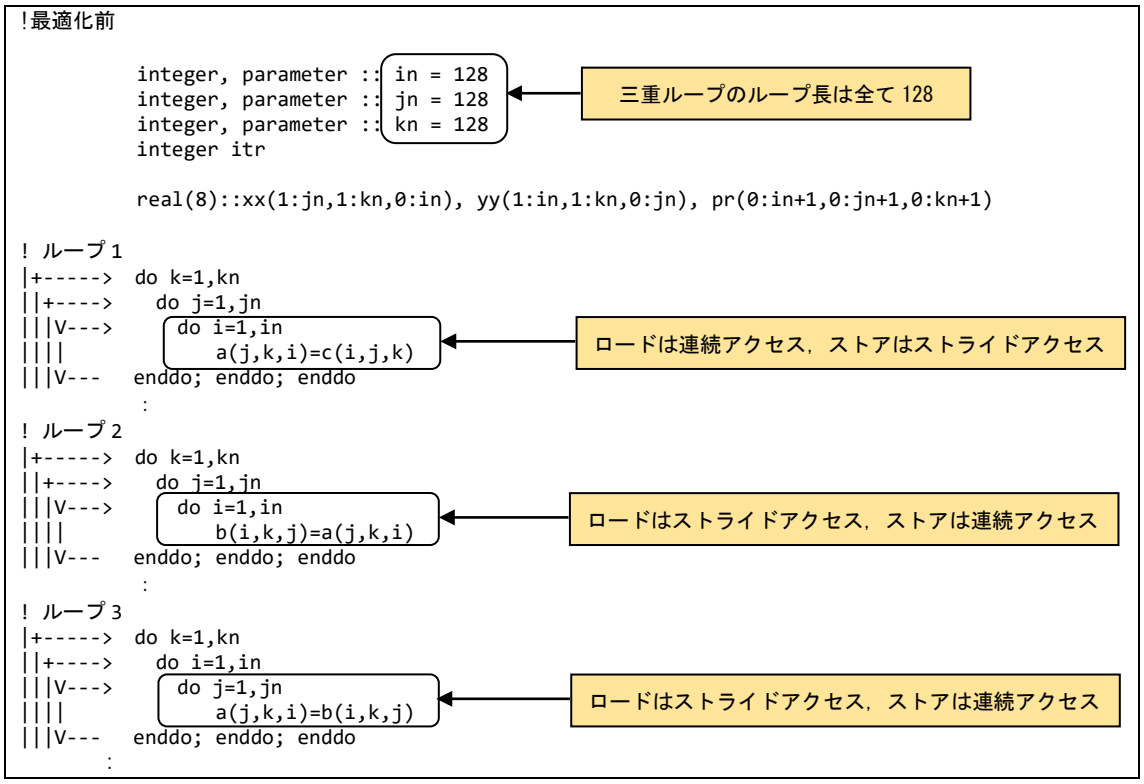

図 6.2.2-1 メモリアクセスの効率化前のコード

(2) 最適化内容

多重ループ内でロード配列, ストア配列へのアクセスパターンが異なる場合には, 以下の点に注意して最適なループ順を検討する.

① ロード, ストアの配列アクセスにおいて, 高次元目をベクトル化の対象としない.

② ストア配列へのアクセスにおいてメモリ負荷が小さくなるようにする.

図 6.2.2-2 に最適化後のコードを示す. まずループ 1 については, ①の条件からロード配列 c, ストア配列 a の 3 次元目がベクトル化の対象とならいようにするため, i および k のループが最内側ループとならないようにする. さらに②の条件からなるべくストア配列の配列アクセスの負荷が小さくなるように考慮すると, ループの順番は外側から i, k, j の順番となる. 同様に, ループ 2, ループ 3 について上記条件を考慮すると, ループ 2 では, 外側から j, i, k, ループ 3 では外側から i, j, k のループの順番が最適と考えられる.

```
!最適化後

          integer, parameter :: in = 128
          integer, parameter :: jn = 128
          integer, parameter :: kn = 128
          integer itr

          real(8)::a(1:jn,1:kn,0:in), b(1:in,1:kn,0:jn), pr(0:in+1,0:jn+1,0:kn+1)

! ループ1
|+----->  do i=1,in
||+---->    do k=1,kn
|||V--->      do j=1,jn
||||              a(j,k,i)=c(i,j,k)
|||V---    enddo; enddo; enddo
           :
! ループ2
|+----->  do j=1,jn
||+---->    do i=1,in
|||V--->     do k=1,kn
||||              b(i,k,j)=a(j,k,i)
|||V---    enddo; enddo; enddo
           :
! ループ3
|+----->  do i=1,in
||+---->    do j=1,jn
|||V--->      do k=1,kn
||||              a(j,k,i)=b(i,k,j)
|||V---    enddo; enddo; enddo
       :
```

図 6.2.2-2 メモリアクセスの効率化後のコード

(3) 性能分析

図 6.2.2-3 に最適化前後の性能値を示す. ループ順を入れ替えたすべてのパターンの性能値を記載しており, PROC.NAME に記載された ijk の順番がループ順をあらわす. 例えば loop1-kji は, 最外側ループが k のループ, 最内側ループが i のループのように, 左に行くほど外側のループであることを示している.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
!ループ1											
10000	**21.251**(6.9)	2.125	2090.1	0.0	94.43	128.0	21.241	0.014	12.178	2.06	**loop1-kji**
10000	18.722(6.0)	1.872	2381.2	0.0	94.08	128.0	18.710	0.016	11.651	22.25	loop1-jki
10000	8.631(2.8)	0.863	5202.9	0.0	93.40	128.0	8.625	0.012	6.800	59.73	loop1-jik
10000	7.366(2.4)	0.737	6097.0	0.0	93.40	128.0	7.358	0.007	6.690	93.96	loop1-ijk
10000	4.164(1.3)	0.416	10668.0	0.0	94.43	128.0	4.150	0.014	1.228	39.46	loop1-kij
10000	**1.718**(0.6)	0.172	25948.0	0.0	94.08	128.0	1.709	0.012	1.149	94.11	**loop1-ikj**
!ループ2											
10000	**36.439**(11.8)	3.644	1219.0	0.0	94.43	128.0	36.444	0.000	12.226	93.02	**loop2-kji**
10000	23.109(7.5)	2.311	1922.1	0.0	94.43	128.0	23.095	0.017	12.212	13.64	loop2-kij
10000	19.369(6.3)	1.937	2301.7	0.0	94.08	128.0	19.367	0.012	11.652	24.68	loop2-ikj
10000	17.839(5.8)	1.784	2499.0	0.0	94.08	128.0	17.835	0.005	10.681	89.30	loop2-jki
10000	12.877(4.2)	1.288	3462.1	0.0	94.08	128.0	12.871	0.008	11.144	75.12	loop2-ijk
10000	**11.939**(3.9)	1.194	3734.1	0.0	94.08	128.0	11.930	0.008	11.021	90.82	**loop2-jik**
!ループ3											
10000	**36.739**(11.9)	3.674	1209.0	0.0	94.43	128.0	36.732	0.007	12.233	92.97	**loop3-kij**
10000	23.740(7.7)	2.374	1871.0	0.0	94.43	128.0	23.742	0.006	12.201	6.34	loop3-kji
10000	19.278(6.2)	1.928	2312.5	0.0	94.08	128.0	19.263	0.020	11.690	33.33	loop3-jki
10000	17.714(5.7)	1.771	2516.6	0.0	94.08	128.0	17.706	0.008	10.687	89.97	loop3-ikj
10000	12.800(4.1)	1.280	3482.7	0.0	94.08	128.0	12.785	0.020	11.139	74.65	loop3-jik
10000	**11.939**(3.9)	1.194	3734.1	0.0	94.08	128.0	11.932	0.007	11.021	91.20	**loop3-ijk**

図 6.2.2-3 メモリアクセスの効率化前後の FTRACE 情報

すべてのループにおいて，最適化後の性能値が最も高くなっていることがわかる．特にループ 1 では実行時間が 21.25 秒から 1.72 秒と約 12 倍短縮されている．

ただし，ループ長の違いやループ内の配列の数の違いなどにより，キャッシュの動作が変化するとメモリアクセス性能も影響を受ける．つまり，本事例で検討した最適化の方は，必ずしもどんなパターンにおいても成立するとは限らない点には注意が必要である．あくまで多重ループのループ順番最適化の参考情報として活用してほしい．

6.2.3. ループアンロールによる高速化

(1) 最適化方針

多重 DO ループでステンシル計算（ストアとロードに配列のインデックスが近い値が用いられる時間発展型の演算）となっている場合，多重ループのアウターアンロールによりメモリアクセス数の削減を図れることが多い．

本事例は outerloop_unroll(n) 指示行の追加により多重ループのアンローリングを行うことでメモリアクセス数の低減を行う手法である．図 6.2.3-1 に最適化前のコードを示す．配列 a の演算において，最内ループは 3 次元目の i のインデックスでベクトル化されている．その外側のループは 2 次元目の j のインデックスの演算を行い，j-1，j，j+1 がロードされ，j にストアされるステンシル計算となっている．

図 6.2.3-2 に示した FTRACE 情報から，ベクトル演算率，平均ベクトル長ともに高い値となっており，ベクトルプロセッサの性能を十分に発揮したプログラムであることが分かる．また REQ.ST B/F の値が 3.20，REQ.LD B/F の値が 6.40 と高く，高いメモリアクセス性能を必要とするプログラムであることが分かる．

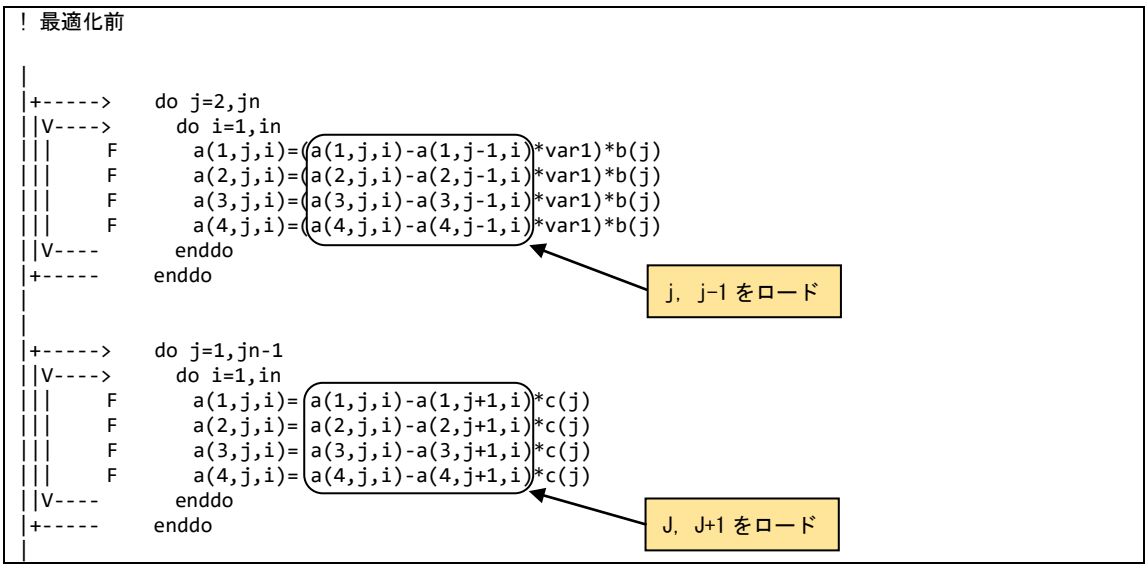

図 6.2.3-1 ループアンロール前のコード

```
FREQUENCY  EXCLUSIVE       AVER.TIME   MOPS  MFLOPS   V.OP   AVER.    VECTOR ..  REQ.   REQ.ST  REQ.LD  ACT.VLD  FLOP COUNT     FMA ELEM.  PROC.NAME
           TIME[sec](  %)   [msec]                    RATIO  V.LEN      TIME ..   B/F      B/F     B/F      B/F

   10000   20.926(100.0)     2.093  2766.9  1247.9  99.22  256.0     20.964     9.60    3.20    6.40     0.00  26112000000  10444800000【最適化前】
```

図 6.2.3-2 ループアンロール前の FTRACE 情報

(2) 最適化内容

図 6.2.3-3 に最適化後のコードを示す. アウターアンロールを行うインデックス j の DO ループの前に指示行 outerloop_unroll(n) を追加することで, インデックス j のループにアンロールされたことを示す ”U” が記載されたことが確認できる. アンロール段数は n を超えない最大の 2 のべき乗数となる. 本事例では n に 2～16 の値を指定して実行し, 最も実行時間の短縮された n=4 を採用した.

```
! 最適化後

|         !NEC$ outerloop_unroll(4)   <-- 指示行により, j のループを 4 段アンロール
|U----->    do j=2,jn
||V---->     do i=1,in
|||     F      a(1,j,i)=(a(1,j,i)-a(1,j-1,i)*var1)*b(j)
|||     F      a(2,j,i)=(a(2,j,i)-a(2,j-1,i)*var1)*b(J)
|||     F      a(3,j,i)=(a(3,j,i)-a(3,j-1,i)*var1)*b(J)
|||     F      a(4,j,i)=(a(4,j,i)-a(4,j-1,i)*var1)*b(J)
||V----      enddo
|U-----    enddo
|
|         !NEC$ outerloop_unroll(4)   <-- 指示行により, j のループを 4 段アンロール
|U----->    do j=1,jn-1
||V---->     do i=1,in
|||     F      a(1,j,i)= a(1,j,i)-a(1,j+1,i)*c(j)
|||     F      a(2,j,i)= a(2,j,i)-a(2,j+1,i)*c(j)
|||     F      a(3,j,i)= a(3,j,i)-a(3,j+1,i)*c(j)
|||     F      a(4,j,i)= a(4,j,i)-a(4,j+1,i)*c(j)
||V----      enddo
|U-----    enddo
|
```

図 6.2.3-3 ループアンロール後のコード

(3) 性能分析

図 6.2.3-4 に最適化前後の性能値を示す.

```
FREQUENCY  EXCLUSIVE       AVER.TIME   MOPS  MFLOPS   V.OP   AVER.    VECTOR ..  REQ.   REQ.ST  REQ.LD  ACT.VLD  FLOP COUNT     FMA ELEM.  PROC.NAME
           TIME[sec](  %)   [msec]                    RATIO  V.LEN      TIME ..   B/F      B/F     B/F      B/F

   10000  20.926(100.0)     2.093  2766.9  1247.9  99.22  256.0     20.964     9.60    3.20    6.40     0.00  26112000000  10444800000【最適化前】
   10000  15.532(100.0)     1.553  3224.6  1681.2  99.24  256.0     15.580     7.25    3.21    4.04     0.00  26112000000  10444800000【最適化後】
```

図 6.2.3-4 ループアンロール前後の FTRACE 情報

outerloop_unroll(n) 指示行により外側ループをアンロールしたことでメモリ負荷が下がり, REQ.LD B/F(Required Load B/F)が 6.40 から 4.04 に減少している. 必要とされるメモリロードが減ることでメモリアクセス時間が減少した結果, 実行時間が 20.9 秒から 15.6 秒に短縮されている.

6.2.4. メモリ割当回数低減による高速化

(1) 最適化方針

図 6.2.4-1 に最適化前のソースコードを示す. 本事例は, 粒子法の一種である Smoothed Particle Hydrodynamics (SPH 法)に基づくミニアプリケーションである, SPH-EXA (https://github.com/unibas-dmi-hpc/SPH-EXA) より抜粋したものである. 本コード片は粒子を MPI プロセスに割り当てるため, 各粒子がシミュレーション空間内のどの領域に存在するか計算する, distributeParticles 関数の一部である.

このコードでは, 各領域に属する粒子 ID のリストである cellList を粒子の座標に応じて構築している. cellList[l]は, l 番目の領域に属する粒子 ID のリストである. 各領域内の粒子数は可変のため, vector<T>の末尾に要素を追加する関数 vector<T>::push_back を使用し, 逐次的に cellList を構築している. しかし, push_back は動的にメモリ領域の拡張をするため, 要素を追加するたび境界チェックを行うため, ベクトル化を阻害してしまう. そこで cellList[l]の長さ, すなわち領域内の粒子数を事前に計算し, メモリ領域を確保しておくことで, push_back の呼び出しを回避する.

図 6.2.4-2 に FTRACE 情報を示す. ベクトル化率が非常に低いことが明らかである. また, Reallocate という, メモリの再割当を行うと推定される関数が 1,200 万回以上も呼び出されていることもわかった.

```
! 最適化前

+-->   for (unsigned int i = 0; i < clist.size(); i++)
|      {
|   I      T xx = std::max(std::min(x[clist[i]], globalBBox.xmax), globalBBox.xmin);
|   I      T yy = std::max(std::min(y[clist[i]], globalBBox.ymax), globalBBox.ymin);
|   I      T zz = std::max(std::min(z[clist[i]], globalBBox.zmax), globalBBox.zmin);
|
|          T posx = normalize(xx, globalBBox.xmin, globalBBox.xmax);
|          T posy = normalize(yy, globalBBox.ymin, globalBBox.ymax);
|          T posz = normalize(zz, globalBBox.zmin, globalBBox.zmax);
|
|          int hx = posx * nX;
|          int hy = posy * nY;
|          int hz = posz * nZ;
|
|   I      hx = std::min(hx, nX - 1);
|   I      hy = std::min(hy, nY - 1);
|   I      hz = std::min(hz, nZ - 1);
|
|          unsigned int l = hz * nX * nY + hy * nX + hx;
|
|   I      cellList[l].push_back(clist[i]);
+--    }
```

図 6.2.4-1 リサイズ回数低減前のソースコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
2	33.215(6.4)	16607.389	898.2	10.0	**0.03**	6.6	0.089	13.083	0.000	100.00	【最適化前】

図 6.2.4-2 リサイズ回数低減前の FTRACE 情報

(2) 最適化内容

図 6.2.4-3 に 最適化後のソースコードを示す. push_back 関数の呼び出しを避けるため, 次の 3 ステップに分けて処理を実現する. すなわち, (1) cellList の各要素のリストの長さを事前に計算 (2) cellList の各要素のリストをリサイズしメモリ領域を確保 (3) cellList の各要素のリストに粒子 ID を代入, である.

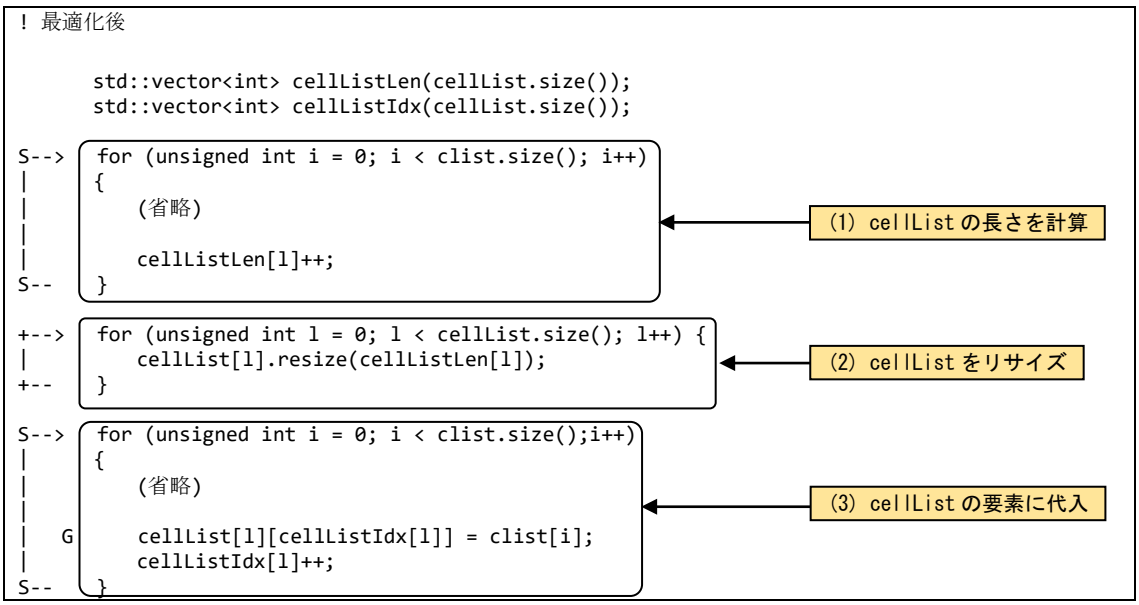

図 6.2.4-3 リサイズ回数低減後のソースコード

(3) 性能分析

図 6.2.4-4 に最適化後の FTRACE 情報を示す. また, 図 6.2.4-5 にリサイズ回数低減前後の Reallocate 関数の実行回数の差異を記載する.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
2	**33.215**(6.4)	16607.389	898.2	10.0	**0.03**	6.6	0.089	13.083	0.000	100.00	【最適化前】
2	**2.697**(0.6)	1348.340	1732.2	206.2	**35.29**	191.5	0.061	0.747	0.000	13.92	【最適化後】

図 6.2.4-4 リサイズ回数低減前後の FTRACE 情報

	最適化前	最適化後
Reallocate 関数の実行回数	1,200 万回程度	440 万回程度

図 6.2.4-5 リサイズ回数低減前後の Reallocate 関数の実行回数

最適化により, 実行時間が 33.3 秒から 2.7 秒に短縮されている. また, 部分ベクトルによってベクトル化率が 0.03% から 35.29% に向上した. また, Reallocate 関数の実行回数が 1,200 万回程度から 440 万回程度まで削減されていることがわかる.

6.3. ファイル I/O 高速化の事例

6.3.1. ファイル I/O の高速化

(1) 最適化方針

AOBA-A でサイズの大きいデータの入出力を行う場合は I/O 時間短縮のため，書式なし記録（バイナリファイル）を行うことが推奨される．これは，書式付記録（テキストファイル）で I/O を行った場合，データの書き出し要素毎に書式変換のためのスカラ処理が必要となるため，著しく実行時間が増加するためである．

本事例は書式なし記録の場合でも，WRITE/READ 文における配列のアクセス方法がスカラ方式の場合には入出力の時間が増加するため，これを配列範囲の指定に変更しベクトル演算とすることで計算時間の短縮を図る．また，Fortran2003 から実装されたファイル I/O の方式で C 言語とも互換性のある，ストリームファイルの書式なし記録で I/O を行った場合の性能も示す．

FTRACE 情報の取得の際に，環境変数 VE_FORT_FILEINF=DETAIL を指定して実行する事で，ファイル I/O に関連する情報（File Information）を採取する事が可能である．図 6.3.1-1 に最適化前のコードを示す．倍精度実数で定義された 3 つの 3 次元配列の WRITE および READ を行い，一度の I/O におけるデータサイズは 402,653,184byte（384MB）となる．外部ファイルの接続方法は順編成ファイルの書式なし記録（unformatted, sequential）である．

図 6.3.1-2 に最適化前の FTRACE 情報を，図 6.3.1-3 に最適化前の File I/O 情報の一部を示す．ファイルは書式なし記録で行われているが，WRITE/READ 文での配列のアクセス方法がスカラ方式となるため，WRITE/READ ともにベクトル演算率が低く，その結果ファイル転送速度（Transfer Rate）は WRITE で約 150,959KByte/sec，READ で約 78,337KByte/sec となっている．

```
! 最適化前

          open(10,file='data.dat',form='unformatted',status='replace')
+------>    write(10) (((a(i,j,k),b(i,j,k),c(i,j,k),i=1,in),j=1,jn),k=1,kn)
          close(10)

          open(10,file='data.dat',form='unformatted',status='old')
+------>    read(10) (((a(i,j,k),b(i,j,k),c(i,j,k),i=1,in),j=1,jn),k=1,kn)
          close(10)
```

図 6.3.1-1 最適化前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
1	4.782(66.9)	4782.318	2650.2	0.0	**0.79**	256.0	0.002	0.029	0.000	100.00	【最適化前・READ】
1	2.364(33.1)	2364.459	2722.3	0.0	**4.69**	255.7	0.005	0.043	0.000	99.96	【最適化前・WRITE】

図 6.3.1-2 最適化前の FTRACE 情報

```
【最適化前・WRITE】(抜粋)
                                Total(In/Out)          Input          Output
Total Data Size (Byte)    :       402653184,              0,       402653184
Max   Data Size (Byte)    :                               0,       402653184
Min   Data Size (Byte)    :                               0,       402653184
Ave   Data Size (Byte)    :       402653184,              0,       402653184
Transfer Rate (KByte/sec) :       150959.571,         0.000,       150959.571

                            Total(In/Out/Aux)          Input          Output
Real Time (sec)           :         2.606056,      0.000000,         2.604777
User Time (sec)           :         2.363568,      0.000000,         2.363455


【最適化前・READ】(抜粋)
                                Total(In/Out)          Input          Output
Total Data Size (Byte)    :       402653184,      402653184,               0
Max   Data Size (Byte)    :                       402653184,               0
Min   Data Size (Byte)    :                       402653184,               0
Ave   Data Size (Byte)    :       402653184,      402653184,               0
Transfer Rate (KByte/sec) :       78337.021,      78337.021,           0.000

                            Total(In/Out/Aux)          Input          Output
Real Time (sec)           :         5.019925,      5.019542,         0.000000
User Time (sec)           :         4.781595,      4.781551,         0.000000
```

図 6.3.1-3 最適化前の File I/O 情報

(2) 最適化内容

図 6.3.1-4 に最適化後のコードを示す. WRITE/READ 文において配列のアクセス方法を配列範囲を指定する方法に変更している. また, 図 6.3.1-5 は accsess='stream' を指定してストリーム方式でのファイル I/O を行うコードである.

```
! 最適化後

          open(10,file='data.dat',form='unformatted',status='replace')
          write(10) a(1:in,1:jn,1:kn),b(1:in,1:jn,1:kn),c(1:in,1:jn,1:kn)
          close(10)

          open(10,file='data.dat',form='unformatted',status='old')
          read(10) a(1:in,1:jn,1:kn),b(1:in,1:jn,1:kn),c(1:in,1:jn,1:kn)
          close(10)
```

図 6.3.1-4 最適化後のコード

```
! ストリーム方式

          open(10,file='data.dat',form='unformatted',status='replace',access='stream')
          write(10) a(1:in,1:jn,1:kn),b(1:in,1:jn,1:kn),c(1:in,1:jn,1:kn)
          close(10)

          open(10,file='data.dat',form='unformatted',status='old',access='stream')
          read(10) a(1:in,1:jn,1:kn),b(1:in,1:jn,1:kn),c(1:in,1:jn,1:kn)
          close(10)
```

図 6.3.1-5 access='stream'を指定したコード

(3) 性能分析

図 6.3.1-6 に最適化前後およびストリーム方式でのファイル I/O を行った場合の FTRACE 情報を，図 6.3.1-7 に最適化後の File I/O 情報，図 6.3.1-8 にストリーム方式での File I/O 情報を示す．

最適化後はファイル I/O がベクトル演算で行われたことにより，ベクトル演算率が約 99% まで増加し，計算時間はほぼ 0 秒となった．このときのファイル転送速度は WRITE で約 2,894,875KByte/sec，READ で約 3,579,814KByte/sec となり，最適化前と比較してそれぞれ約 19 倍と約 45 倍の性能向上となった．

ストリーム方式での I/O の場合，ファイル転送速度は WRITE で約 2,952,594KByte/sec，READ で約 9,514,898KByte/sec となり，最適化後と比較して WRITE ではほぼ同等の性能であったが，特に READ では約 2.6 倍のさらなる性能向上がみられた．

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
1	**4.782**(66.9)	4782.318	2650.2	0.0	**0.79**	256.0	0.002	0.029	0.000	100.00	【最適化前･READ】
1	**0.005**(40.7)	4.555	44732.7	0.0	**99.00**	256.0	0.003	0.001	0.000	93.45	【最適化後･READ】
1	**0.003**(39.4)	3.288	30984.1	0.0	**98.80**	255.9	0.001	0.001	0.000	95.31	【ストリーム･READ】
1	**2.364**(33.1)	2364.459	2722.3	0.0	**4.69**	255.7	0.005	0.043	0.000	99.96	【最適化前･WRITE】
1	**0.005**(48.8)	5.456	37374.2	0.0	**98.92**	256.0	0.003	0.001	0.000	0.20	【最適化後･WRITE】
1	**0.004**(46.6)	3.889	26306.2	0.0	**98.66**	256.0	0.002	0.001	0.000	0.26	【ストリーム･WRITE】

図 6.3.1-6 最適化前後の FTRACE 情報

```
【最適化後・WRITE】(抜粋)
                                Total(In/Out)          Input          Output
Total Data Size (Byte)     :       402653184,              0,       402653184
Max   Data Size (Byte)     :                               0,       402653184
Min   Data Size (Byte)     :                               0,       402653184
Ave   Data Size (Byte)     :       402653184,              0,       402653184
Transfer Rate (KByte/sec)  :      2894875.355,         0.000,      2894875.355

                           Total(In/Out/Aux)          Input          Output
Real Time (sec)            :         0.157460,      0.000000,       0.135832
User Time (sec)            :         0.004565,      0.000000,       0.004444


【最適化後・READ】(抜粋)
                                Total(In/Out)          Input          Output
Total Data Size (Byte)     :       402653184,      402653184,               0
Max   Data Size (Byte)     :                       402653184,               0
Min   Data Size (Byte)     :                       402653184,               0
Ave   Data Size (Byte)     :       402653184,      402653184,               0
Transfer Rate (KByte/sec)  :      3579814.437,     3579814.437,          0.000

                           Total(In/Out/Aux)          Input          Output
Real Time (sec)            :         0.110245,      0.109843,       0.000000
User Time (sec)            :         0.003832,      0.003788,       0.000000
```

図 6.3.1-7 最適化後の File I/O 情報

```
【ストリーム・WRITE】(抜粋)
                                Total(In/Out)            Input           Output
Total Data Size (Byte)     :       402653184,               0,        402653184
Max   Data Size (Byte)     :                                0,        402653184
Min   Data Size (Byte)     :                                0,        402653184
Ave   Data Size (Byte)     :       402653184,               0,        402653184
Transfer Rate (KByte/sec)  :      2952594.761,          0.000,       2952594.761

                            Total(In/Out/Aux)            Input           Output
Real Time (sec)            :         0.134867,         0.000000,         0.133176
User Time (sec)            :         0.003024,         0.000000,         0.002914


【ストリーム・READ】(抜粋)
                                Total(In/Out)            Input           Output
Total Data Size (Byte)     :       402653184,       402653184,                0
Max   Data Size (Byte)     :                        402653184,                0
Min   Data Size (Byte)     :                        402653184,                0
Ave   Data Size (Byte)     :       402653184,       402653184,                0
Transfer Rate (KByte/sec)  :      9514898.110,     9514898.110,            0.000

                            Total(In/Out/Aux)            Input           Output
Real Time (sec)            :         0.041941,         0.041326,         0.000000
User Time (sec)            :         0.002582,         0.002539,         0.000000
```

図 6.3.1-8 ストリーム方式での File I/O 情報

6.4. MPI 通信処理改善による高速化の事例

6.4.1. 複数の通信処理をまとめることによる高速化

(1) 最適化方針

本事例は, MPI 通信処理を削減することで高速化を行った例である.

図 6.4.1-1 に最適化前のコードを示す. MPI プロセス毎に変数 var[1-6]が個別に算出され, それぞれについて MPI_Allreduce 通信処理が行われている. その後, 変数 var7 が(MPI_Allreduce 通信によって得られた)全 MPI プロセス合計の var[1-6]の総和として算出されている. 後続の処理で参照されるのは var7 のみである.

MPI プロセス毎に var[1-6]の総和を算出し, その値について MPI_Allreduce 通信を行うことで MPI 通信回数を削減することができる.

図 6.4.1-2 に示す最適化前の FTRACE 情報では, 10,800 回の MPI 通信によって合計 84.4 kByte が MPI 通信の対象となっている. MPI 通信回数を削減することで性能向上が期待できる.

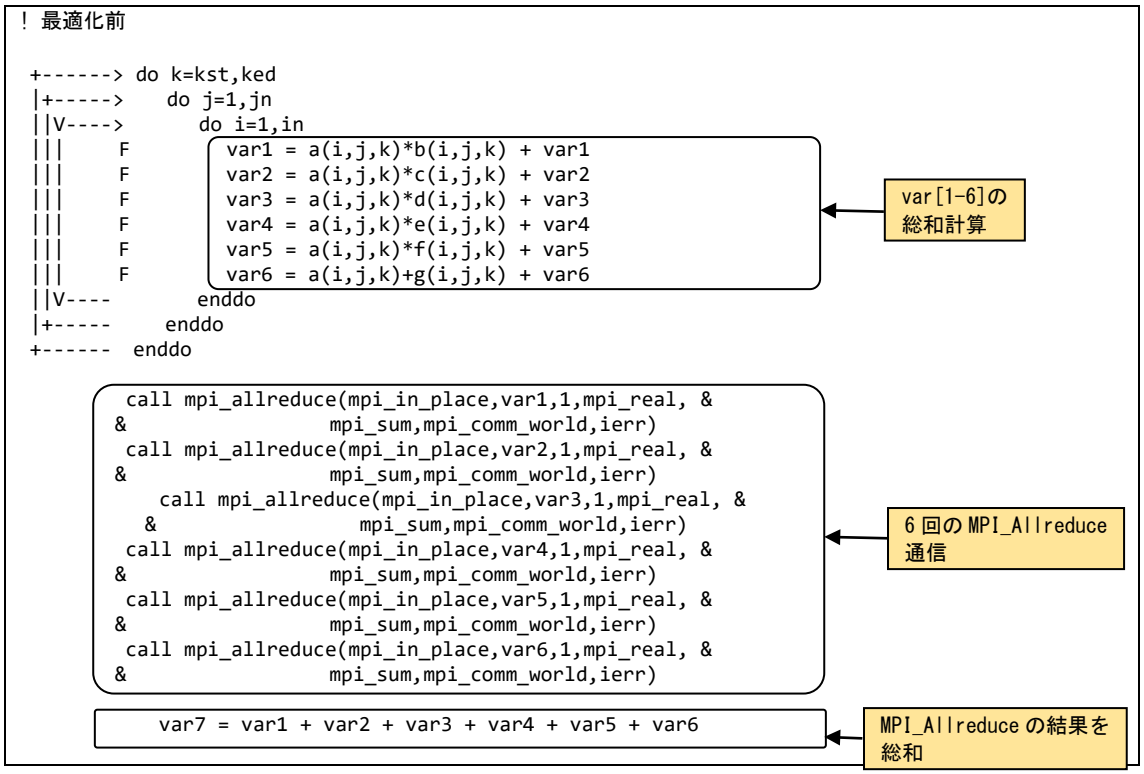

図 6.4.1-1 最適化前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
1800	1.957(0.6)	1.087	146231.9	70243.6	99.53	207.7	1.913	0.019	0.000	0.14	【最適化前】

ELAPSED TIME[sec]	COMM.TIME [sec]	COMM.TIME / ELAPSED	IDLE TIME [sec]	IDLE TIME / ELAPSED	AVER.LEN [byte]	COUNT	TOTAL LEN [byte]	PROC.NAME
1.958	0.048	0.024	0.012	0.006	8.0	**10800**	**84.4K**	【最適化前】

図 6.4.1-2 最適化前の FTRACE 情報

(2) 最適化内容

図 6.4.1-3 に最適化後のコードを示す. MPI プロセス毎に var[1-6]の総和を算出し, その値について MPI_Allreduce 通信を行うことで MPI 通信回数を削減することができる.

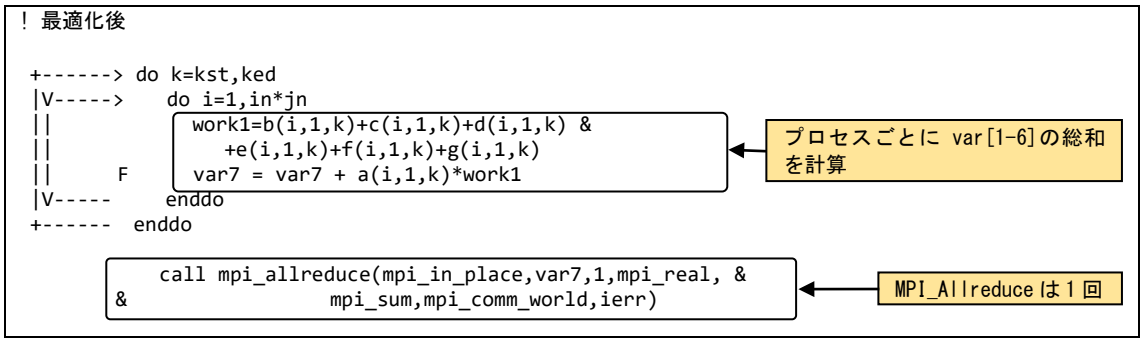

図 6.4.1-3 最適化後のコード

(3) 性能分析

図 6.4.1-4 に最適化前後の性能値を示す.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
1800	**1.957**(0.6)	1.087	146231.9	70243.6	99.53	207.7	1.913	0.019	0.000	0.14	【最適化前】
1800	**1.460**(1.7)	0.811	102840.0	13669.9	99.66	255.8	1.444	0.013	0.000	0.00	【最適化後】

ELAPSED TIME[sec]	COMM.TIME [sec]	COMM.TIME / ELAPSED	IDLE TIME [sec]	IDLE TIME / ELAPSED	AVER.LEN [byte]	COUNT	TOTAL LEN [byte]	PROC.NAME
1.958	0.048	0.024	0.012	0.006	8.0	**10800**	**84.4K**【最適化前】	
1.461	0.012	0.008	0.002	0.001	8.0	**1800**	**14.1K**【最適化後】	

図 6.4.1-4 最適化前後の FTRACE 情報

MPI 通信回数の削減により, MPI 通信回数が 10,800 回から 1,800 回に, MPI 通信量が 84.4 kByte から 14.1 kByte に削減されている. 結果として実行時間が 2.0 秒から 1.5 秒に短縮されている.

6.4.2. 演算と通信のオーバーラップによる高速化

(1) 最適化方針

MPI の通信処理時間を削減する手法として, 演算処理とのオーバーラップがある. 演算処理と同時に MPI 通信処理を実行することで, 通信処理時間を隠蔽するものである. この手法の適用可能な条件として, 通信処理に依存しない(直接関係のない)演算処理が存在する必要がある. 例えば, 差分計算を MPI 化する場合, 各 MPI プロセスに割り当てられた領域のうち, 境界領域については, 隣接する MPI プロセスとデータ交換が必要になる. この場合, データ交換が必要な境界領域を先に演算し, データ交換の必要ない境界領域以外の演算中にデータ交換(MPI 通信処理)を行うことで, 演算と通信のオーバーラップを実現可能である.

本高速化では演算と通信のオーバーラップの手法として, スレッド操作を用いる. MPI 並列とスレッド並列を組み合わせたハイブリッド MPI 実行(VE 間は MPI 並列, VE 内はスレッド並列)において, 演算処理に必要なデータのメモリからの供給が間に合わず, 演算処理が待ち状態となりスレッド並列の効果が得られない場合がある. このような場合に, 特定のスレッドが MPI 通信処理を担当し, 演算処理を担当する他のスレッドと平行に実行することで演算処理と通信処理のオーバーラップを実現する.

図 6.4.2-1(左)に演算処理と通信処理をオーバーラップさせない場合の処理フローを示す. MPI プロセスに割り当てられた全領域の計算をスレッドに割り当て, 各スレッドが並列に演算処理を実行する. 演算処理が終了すると, 隣接する領域間で境界要素の交換を行うため, 境界要素を収集する処理を実行する. 次に, 収集したデータを隣接する領域を担当する MPI プロセス間で交換するための通信処理を行う. SX-Aurora TSUBASA の MPI は, どのスレッドも MPI 手続きを呼び出すことができるが, 複数のスレッドが同時に MPI 手続きを実行することはできないため(MPI_THREAD_SERIALIZED 機能), 境界領域の通信はマスタースレッド(スレッド 0)が担当して実行する. 通信処理終了後, 通信されたデータをもとの配列に格納する境界要素の展開処理を

実行する. 図 6.4.2-1(右)に, 演算処理と通信処理をオーバーラップさせた場合の処理フローを示す. まず, 境界領域の計算を全スレッドで分担して実行する. 次に, マスタースレッドが境界要素の収集, データ交換のための通信, 交換された境界要素の展開を担当する. そのほかのスレッド(スレッド 1 から 3)は境界領域以外の計算を分担して担当する. このとき, スレッドに対する割り当て方法は, OpenMP が有する schedule 機能を用いて自動で設定する.

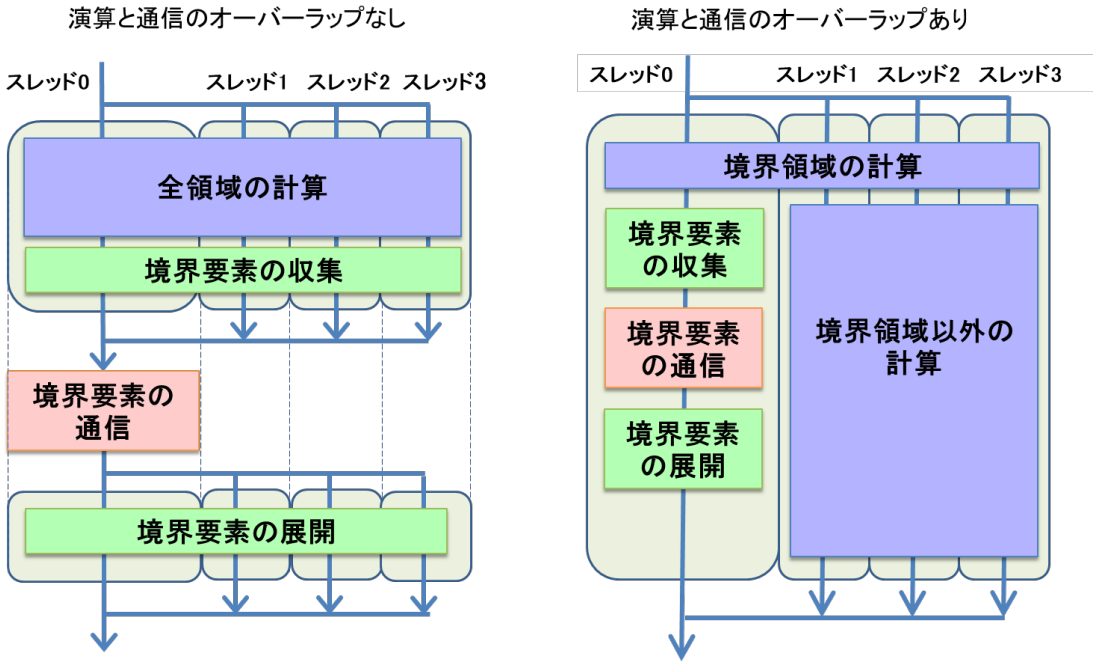

図 6.4.2-1 演算と通信のオーバーラップなし(左)とあり(右)の処理フロー

演算処置と通信処理をオーバーラップさせる手法において, スレッドの割り当て方法には, OpenMP が有する schedule 機能を用いて自動で設定する方法と, ユーザ側が手動で設定する方法がある.

図 6.4.2-2(左)に OpenMP による自動スレッド操作の例を示す. 図中の赤字は OpenMP の指示文を表す. parallel 指示文により, スレッドが生成される. 通信処理(境界要素の収集・通信・展開)の前後に master 指示文と end master 指示文を挿入することで, スレッド 0 が通信処置を担当する. do 指示文の指示句である schedule に dynamic を指定することで, 各スレッドに演算処置が動的に割り当てられる. また, スレッド 0 の通信処理が終了した時点で演算処置が終了していなければ, スレッド 0 にも演算処置が割り当てられる. OpenMP による自動スレッド操作はユーザ側の負担(プログラム修正)が小さいが, OpenMP による操作のためのオーバヘッドが大きくなる問題がある. 図 6.4.2-2(右)に, OpenMP による手動スレッド操作の例を示す. 通信処理をスレッド 0 に割り当てる部分は自動スレッド割り当てと同じであるが, 演算処理の各スレッドへの割り当てを OpenMP の do 指示文を使用せず, DO 文に与える開始値と終了値に事前に計算しておいたスレッドごとの開始値と終了値を指定する. 演算処理はスレッド 0 を除くスレッドで均等に分担し, 均等に分担できなかった場合の、余りの処理をスレッド 0 が担当する. これは, 本手法を適用する前提

として，すべてのスレッドを演算に割り当ててもデータ供給が間に合わないメモリ負荷の高いアプリケーションを想定しているため，スレッド 0 には，通信処理と，最小限の演算処置を割り当てるためである．図 6.4.2-2(右)の例では，MPI プロセスに割り当てられた領域の大きさ(DO ループの繰り返し数)が 100 の場合のスレッドごとの開始値，終了値を示している．スレッド 1～3 の繰り返し数が 33 になるように分割され，あまりの 1 回の処理をスレッド 0 が行うように開始値，終了値が設定されている．このスレッド操作の場合，手動でスレッド操作を行うため OpenMP によるオーバヘッドの影響が小さい．事前に各スレッドに割り当てる演算の開始値と終了値を計算しておく必要があるが，わずか 20 行程度の処理の追加である．

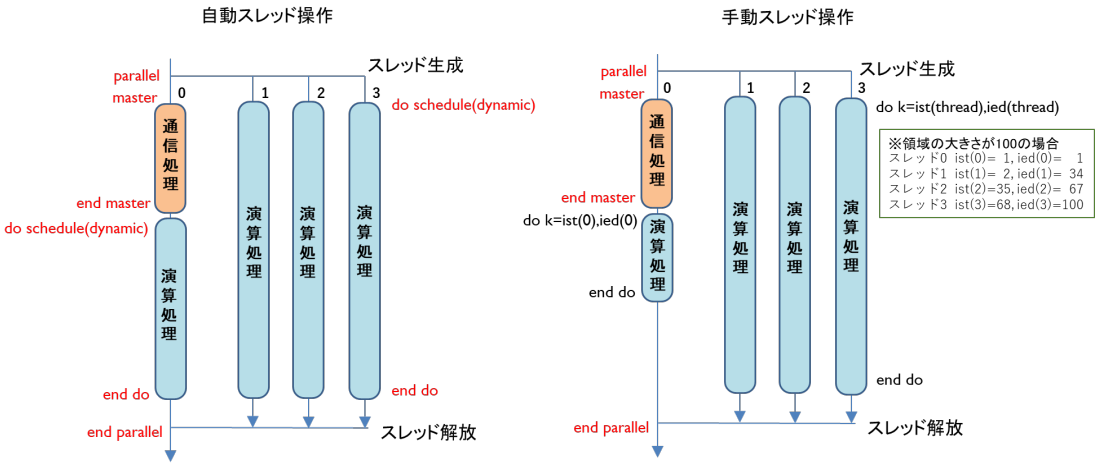

図 6.4.2-2 OpenMP による自動スレッド生成(左)と手動スレッド生成(右)

(2) 最適化内容

演算と通信のオーバーラップによる高速化事例を，Poisson 方程式に対して SOR(Successive Over-Relaxation)法を用いて計算するプログラムを使用して説明する．図 6.4.2-3(左)に演算と通信のオーバーラップ適用前，図 6.4.2-3(右)に適用後の SOR 法の処理フローを示す．

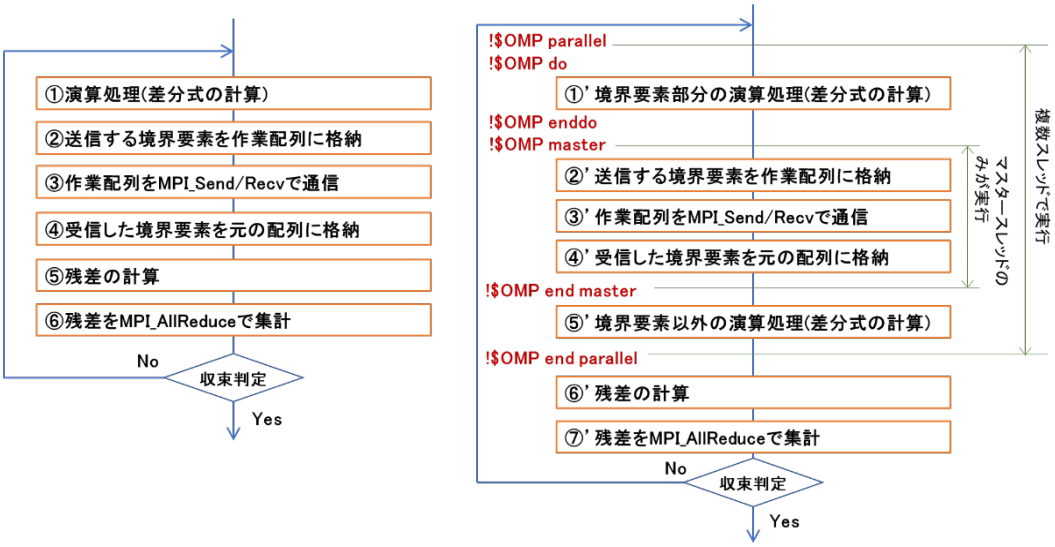

図 6.4.2-3 オーバーラップ適用前(左)と適用後(右)の SOR 法の処理フロー

まず，演算と通信のオーバーラップ前の処理フローでは，①で割り当てられた全領域の計算を行い，②③④では隣接する MPI プロセスが担当する境界要素の値を参照するため，MPI 通信を用いてデータの転送を行う．②でメモリ上連続でないアドレスに格納されている境界要素を送信用の作業配列に連続して格納し，④で受信した作業配列を元の配列のアドレスに格納する処理を行う．⑤⑥で SOR 法の収束判定のための処理を行っている．

次に，演算と通信をオーバーラップ後の処理フローでは，①’の処理で，②’～ ④’の送受信に使用する境界領域の計算を全スレッドで行う．②’から ④’の処理は前後に OpenMP 指示文 master / end master を挿入することで，スレッド 0 のみで実行することを指定する．⑤’の境界要素以外の演算処理の実装方法は図 6.4.2-2 に示した通り，自動スレッド操作と手動スレッド操作で異なる．図 6.4.2-4 に，手動スレッド操作の実装例を示す．OpenMP の指示文 do を用いて，外側の K のループをスレッド並列化の対象にするが，schedule 節を付加することで OpenMP が用意するループのスケジュールを使用する．schedule 節の type に dynamic を指定すると動的なスケジュールが適用される．

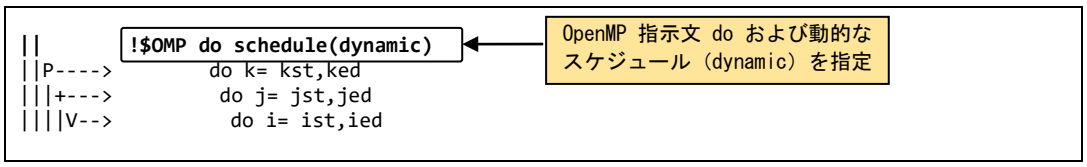

図 6.4.2-4 手動スレッド操作の例

図 6.4.2-5 に手動スレッド操作時のスレッドに割り当てるループの開始値と終了値の計算例を示す．配列 kst2 に各スレッドのループの開始値，配列 ked2 に各スレッドのループの終了値を格納する．配列 kst2，ked2 は動的割り当て配列として，関数 omp_get_num_threads で取得した総スレッド数で配列サイズを定義する．スレッド 1 から最終スレッドにループを均等に割り当てるため，ループに余りが発生する場合にはスレッド 0 に割り当てる． 図 6.4.2-6 に手動スレッド操作時の演算ループに対する処置を示す．手動スレッド操作では，あらかじめ計算しておいたスレッドごとのループの開始値と終了値を使用して各スレッドが担当するループを実行できるようにする．

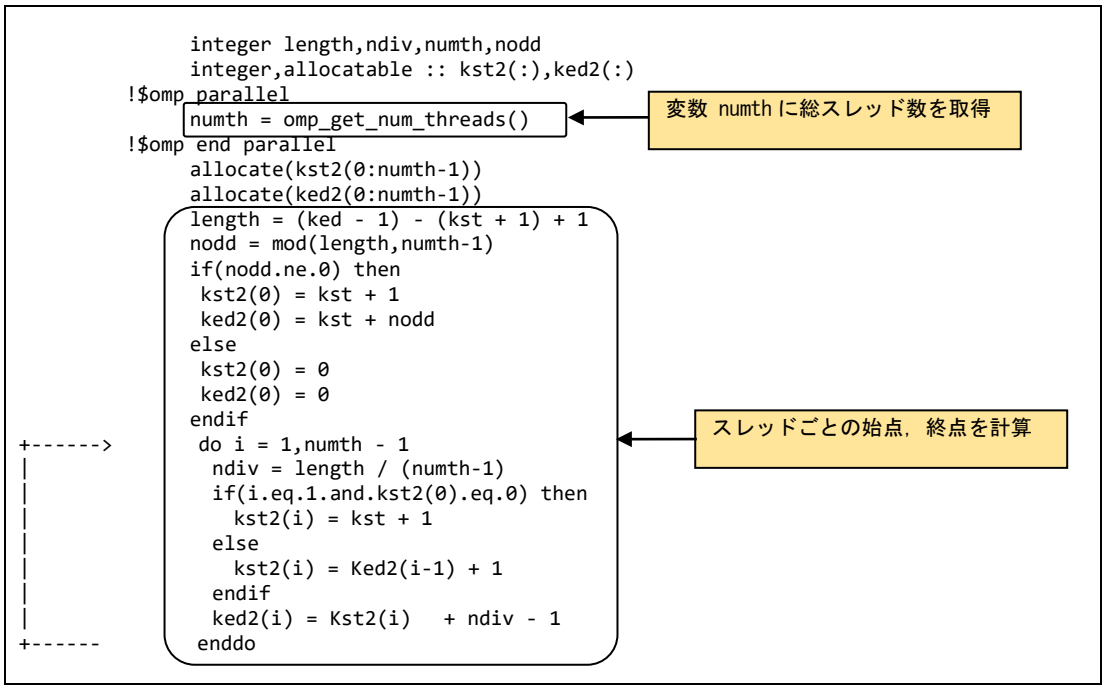

図 6.4.2-5 手動スレッド操作の例(ループの始点・終点を求める)

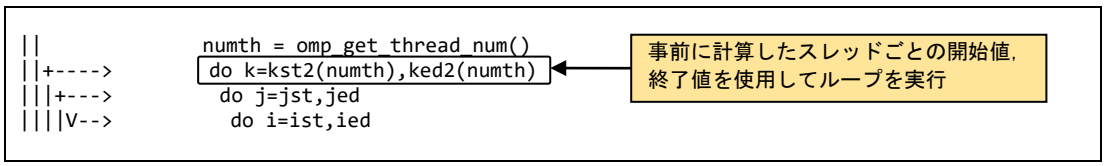

図 6.4.2-6 手動スレッド操作の例(演算ループに対する処置)

(3) 性能分析

図 6.4.2-7 にSOR法ループの測定結果を示す．オーバーラップなしの場合は，通信処置時間（Communucation Time）が 2,059 秒であり，全体時間から通信時間を除いた時間（Calculation Time）は 2,222 秒である．演算と通信のオーバーラップを適用し，かつ演算部分で OpenMP の自動スレッド操作を適用した自動スレッドオーバーラップの場合，全体実行時間が 2,988 秒となり，約 1.5 倍の性能向上が得られた．また，手動スレッド操作による手動スレッドオーバーラップの場合，OpenMP のオーバヘッドが削減されたことにより，自動スレッドオーバーラップよりもさらに実行時間が短縮された．

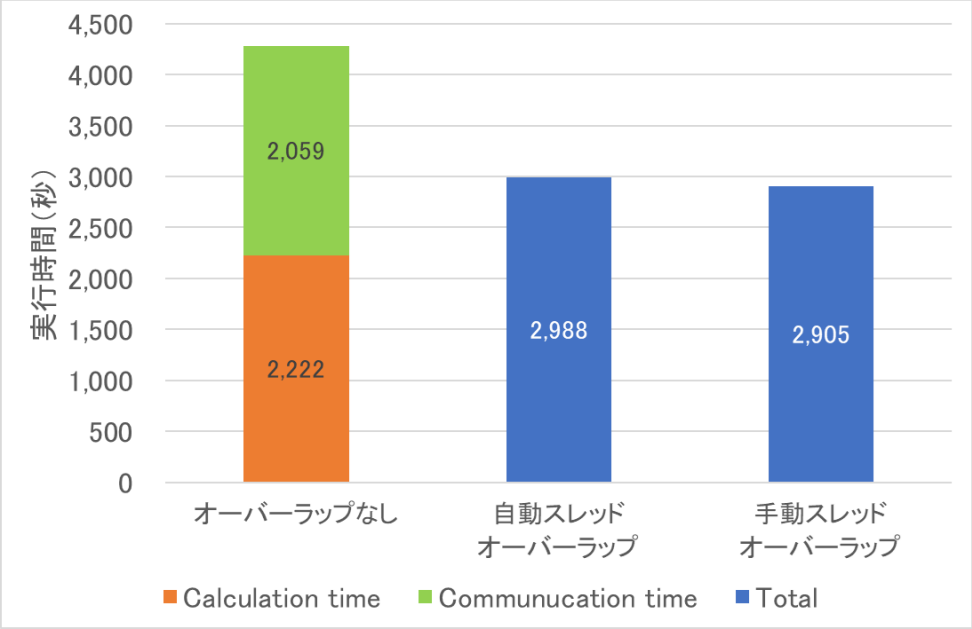

図 6.4.2-7 SOR 法ループの性能測定結果

6.5. VH-VE の連携による高速化の事例

6.5.1. VH Call による高速化(1)

(1) 最適化方針

「5.3.1. VH Call」に記載しているように, VE が不得意とする処理を VH にオフロードすることで性能を向上させることができる. 本事例では, Fortran コードを VH Call で高速化する方法を示す.

図 6.5.1-1 に, 最適化前の FTRACE 情報を示す. また, 図 6.5.1-2 に最もコストの高いサブルーチン SUB_1 の呼び出し構造の概要図を記載する. SUB_1 は, MAIN 関数から呼び出され, SUB_1 自身も別のサブルーチン SUB_A を呼び出している. さらに, SUB_A は SUB_1 にインライン展開される構造をしている. サブルーチン SUB_1 の性能情報(インライン展開された SUB_A の性能情報も含む)をみると, ベクトル演算率, 平均ベクトル長がともに小さく, ベクトル処理に向いていないことがわかる. そこで, SUB_1 配下の処理を VH にオフロードする.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
1001	49.550(22.0)	49.500	838.2	136.0	**22.34**	**2.7**	28.453	15.954	0.000	00.00	SUB_1
:											

図 6.5.1-1 最適化前の FTRACE 情報

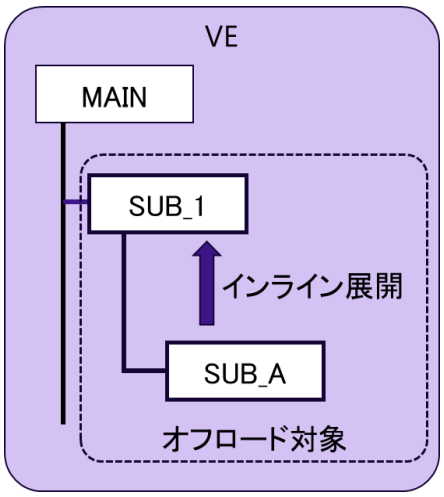

図 6.5.1-2 サブルーチン SUB_1 の呼び出し構造の概要図

(2) 最適化内容

図 6.5.1-3 に，VH Call 適用後の実行イメージ図を記載する．VH にオフロードするサブルーチンを切り出し，共有ライブラリ libvhcal.so としてコンパイルしておく．VE 側のプログラムには，VH Call を利用するための手続き(fvhcall_install，fvhcall_find など)を記載し，fvhcall_invoke_with_args により，VH 側の処理を VE 側から呼び出す．各手続きの概要については，「5.3.1 VH Call」を参照のこと．

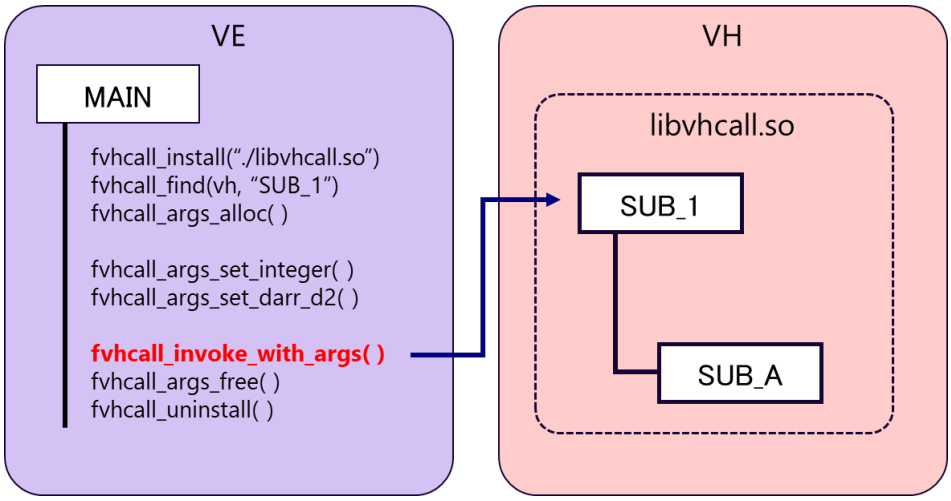

図 6.5.1-3 VHcall 適用後の実行イメージ図

図 6.5.1-4 に共有ライブラリ libvhcall.so のコンパイル方法を記載する．ソースコード vh_fortran.f90 には，SUB_1 および SUB_A を記載する．本事例では，GNU コンパイラを使用して共有ライブラリを作成している．

```
$ gfortran -shared -fPIC -o libvhcall.so vh_fortran.f90
```

図 6.5.1-4 共有ライブラリのコンパイル方法

図 6.5.1-5 に, VE 側で実行するコードのうち, VH Call に関連した部分を抜粋して記載する. まず, Fortran プログラムで VH Call を使用するためには, vhcall_fortran モジュールを参照する必要があるため, use 文で vhcall_fortran モジュール使用を宣言する. そして, fvhcall_install により, 作成した共有ライブラリ libvhcall.so を呼び出す. 次に, vhcall_find により, 共有ライブラリの中からサブルーチン SUB_1 を探す. その後, 手続きに渡す引数を作成するため, fvhcall_args_alloc , fvhcall_args_set_XXXX を実行する. ここで, XXXX の部分は, 引数の型によって変わり, 必要な引数の数だけ記載する必要がある. この事例では, 2 つの単精度整数型の変数, int1 および int2, 倍精度実数型の 2 次元配列 array1, 倍精度実数型の 3 次元配列 array2 の 4 つの引数を設定している. fvhcall_intent_in が, 指定されている場合, その引数が VH 側で実行されるライブラリへの入力として扱われ, fvhcall_intent_out が指定されている場合, その引数は出力となる. 入力, 出力の両方になる場合には, fvhcall_intent_inout を指定する. 引数の設定後, VH 側の手続きを呼びだすために fvhcall_invoke_with_args を実行する. VH 上での処理が完了すると, fvhcall_args_free によって引数を開放し, fvhcall_args_uninstall で共有ライブラリをアンロードする.

```
!VE 側で実行するコード
 :
use vhcall_fortran
implicit none
integer(8) :: vh,sym,ca
integer :: ret
 :
vh = fvhcall_install("./libvhcall.so")
sym = fvhcall_find(vh, "SUB_1")
ca = fvhcall_args_alloc()
 :
ret = fvhcall_args_set_integer(ca, fvhcall_intent_in, 1, int1)
ret = fvhcall_args_set_integer(ca, fvhcall_intent_in, 2, int2)
ret = fvhcall_args_set_darr_d2(ca, fvhcall_intent_in, 3, array1)
ret = fvhcall_args_set_darr_d3(ca, fvhcall_intent_out,4, array2)

ret = fvhcall_invoke_with_args(sym, ca)   <-- VH 側の手続きの呼び出し

call fvhcall_args_free(ca)
ret = fvhcall_uninstall(vh)
 :
```

図 6.5.1-5 VE 側で実行するコード

(3) 性能分析

表 6.5.1-1 に, 最適化前後のサブルーチン SUB_1(SUB_A の実行時間も含む)の実行時間を記載する.

表 6.5.1-1 最適化前後の SUB_1(SUB_A を含む)の実行時間

	最適化前	最適化後
実行時間	49.550 秒	21.257 秒

VH Call により処理を VH 側にオフロードすることで，実行時間が 49.6 秒から 21.3 秒に短縮された．

6.5.2. VH Call による高速化(2)

(1) 最適化方針

「6.5.1. VH Call による高速化(1)」では Fortran プログラムで VH Call を使用した高速化の事例を紹介したが，ここでは C++コードに VHcall を適用した事例を示す．

図 6.5.2-1 に，本プログラムの時間発展ループの概念図を示す．Main 関数内の時間発展ループでは複数の関数が実行されるが，FUNC_2 及び FUNC_3 はベクトル化できない処理を含んでいる．図 6.5.2-2 に FUNC_2，FUNC_3 の実行時間，平均ベクトル長，ベクトル演算率を記載する．FUNC_2 と FUNC_3 を合わせた実行時間は，全体の約 7 割を占めていることがわかる．さらに，これらの処理のベクトル演算率，平均ベクトル長は小さく，VH 側にオフロードして実行することで実行時間の短縮が期待できる． そこで，FUNC2，FUNC3 を VH にオフロードする．

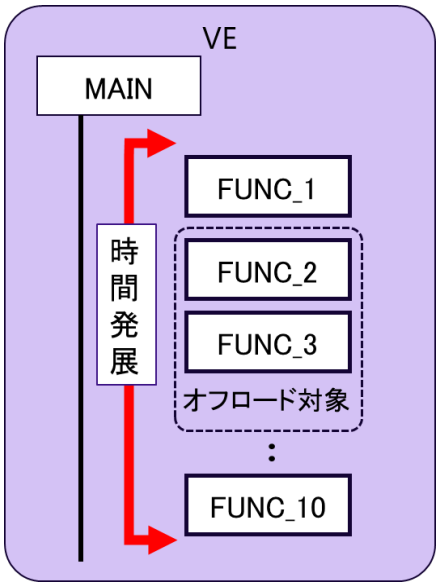

図 6.5.2-1 時間発展ループの概念図

	実行時間	ベクトル演算率	平均ベクトル長
FUNC_2	44.812 秒	63.5 %	174.3
FUNC_3	16.827 秒	0.0 %	0.0
プログラム全体	86.374 秒	-	-

図 6.5.2-2 最適化前の FTRACE 情報

(2) 最適化内容

図 6.5.2-3 に，VH Call 適用後の実行イメージ図を記載する．FUNC_2 内で生成されるデータは，FUNC_3 でしか使用されないため，FUNC_2 と FUNC_3 を 1 つの関数 FUNC_23 としてまとめることで，VE-VH 間の通信量を削減している．VE 側のプログラムでは，VH Call 利用のための前準備（共有ライブラリの呼び出し，引数の作成など）を行い，vhcall_invoke_with_args により，VH 側の処理を呼び出す．

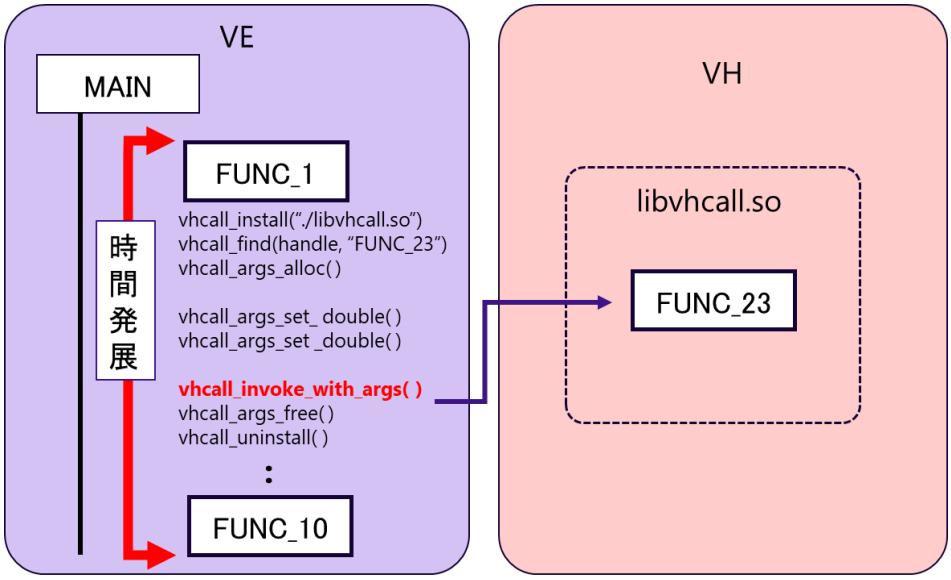

図 6.5.2-3 VHcall 適用後の実行イメージ図

Fortran プログラムの VHcall と同様に，C++プログラムにおいても，VH で利用可能なコンパイラを使用して共有ライブラリのコンパイルを行う．図 6.5.2-4 に，共有ライブラリ libvhcall.so のコンパイル方法を記載する．vh.cpp には，FUNC_2，FUNC3 の処理をまとめた FUNC_23 が記載されている．本事例では，GNU コンパイラを使用して共有ライブラリを作成している．

```
$ g++ -Ofast -march=native -fopenmp -I include -shared -o libvh.so -std=c++11 -fPIC vh.cpp
```

図 6.5.2-4 共有ライブラリのコンパイル方法

図 6.5.2-5 に，VE 側で実行するコードのうち，VH Call に関連した部分を抜粋して記載する．まず，C/C++プログラムで VHcall を使用するために，#include ディレクティブにより，ヘッダファイル libvhcall.h を読み込む．そして，vhcall_install により，作成した共有ライブラリ libvhcall.so を呼び出す．次に，vhcall_find により，共有ライブラリの中から関数 FUNC_23 を探す．関数への引数の引き渡しは，vhcall_args_alloc，vhcall_args_set_XXXX により行われる．ここで，XXXX は引数により異なり，記載された例では，倍精度浮動小数点型の変数 double1，4 バイト符号付き整数型の変数 int1，4 バイト符号なし整数型の変数 u_int1，8 バイト符号なし整数型の変数 u_long1，ポインタ ptr1 となっている．vhcall_args_set_pointer で指定されている VHCALL_INTENT_OUT は，その引数が，VH 側で実行される関数からの出力であることをあらわす．引数が VH 側で実行される関数への入力になる場合には，VHCALL_INTENT_IN を，入出力になる場合には，

VHCALL_INTENT_INOUT を指定する. 引数の設定後, VH 側の関数を呼び出すために vhcall_invoke_with_args を実行する. VH 上での処理が完了すると vhcall_args_free によって引数を開放し, vhcall_args_uninstall で共有ライブラリをアンロードする.

```
!VE 側で実行するコード
  :
#include <libvhcall.h>
  :
    static vhcall_handle handle = vhcall_install("./libvhcall.so");
    static int64_t symid = vhcall_find(handle, "FUNC23");

    int ret;

    vhcall_args *ca  = vhcall_args_alloc();

    ret = vhcall_args_set_double(ca, 0, double1);
      :
    ret = vhcall_args_set_i32(ca, 6, int1);
      :
    ret = vhcall_args_set_u32(ca, 9, u_int1);
      :
    ret = vhcall_args_set_u64(ca, 10, u_long1);
      :
    ret = vhcall_args_set_pointer(ca, VHCALL_INTENT_OUT, 19, ptr1, sizeof(int) * n);
      :
    ret = vhcall_invoke_with_args(symid, ca, NULL);
      :
    vhcall_args_free(ca);
    vhcall_uninstall(handle);
}

 :
```

図 6.5.2-5 VE 側で実行するコード

(3) 性能分析

表 6.5.2-1 に, 最適化前後の関数 FUNC_1 および FUNC_2 の実行時間を記載する.

表 6.5.2-1 最適化前後の FUNC_1 および FUNC_2 の実行時間

	最適化前	最適化後
実行時間(FUNC_1 + FUNC_2)	44.812 秒	19.005 秒

VH Call により処理を VH 側にオフロードすることで, 実行時間が 44.812 秒から 19.005 秒に短縮された.

6.5.3. VH-VE Hybrid MPI による高速化

(1) 最適化方針

物理現象の時間発展の様子を計算するシミュレーションでは, 特定のタイムステップごとに, 計算結果をファイルに出力し, 後処理において時間変化の様子を確認することがよくある. ファイル出力の間隔を短くすればするほど, 現象の時間発展の様子をより短い時間間隔で解明することが

できる. しかし, ファイル出力の回数が増える分, ファイル I/O のコストも増加する. 特に, 1 回あたりの出力ファイルのサイズが大きい場合には, プログラム全体に占める I/O 時間の割合が無視できないくらいに大きくなる場合もある. 本事例は, VE-VH Hybrid MPI により, ファイル I/O にかかる時間を削減した事例である.

本事例のプログラムは, 自然現象の時間発展の様子をリアルタイムに予測計算するものであり, 入力データを一定の時間間隔で受信するという特徴がある. つまり, 次の入力データを受信するまでに, シミュレーションを完了させなければならないという制約がある.

図 6.5.3-1 に本プログラムの全体概要図を記載する. 入力データの受信インターバルは 30 分であり, この間に, 7 時間分の物理シミュレーションを実行する. オリジナルの実行条件では, 1 時間分のシミュレーションごとにファイル出力が行われ, 1 回あたり 160MB のファイルを 4 ファイル出力する. 図 6.5.3-2 は, 最適化前のプログラム全体の実行時間を示している. 1 時間に 1 回ファイル出力を行うケースで, すでに制限時間の 30 分ぎりぎりの値となっており. ファイルの出力間隔を短くすると制限時間を超過してしまうことがわかる.

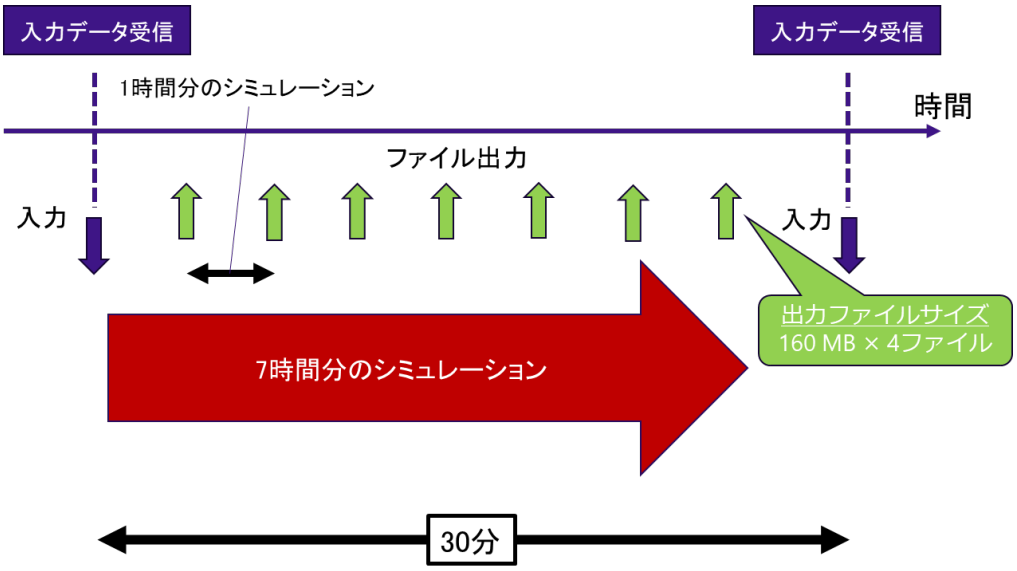

図 6.5.3-1 プログラムの全体概要図

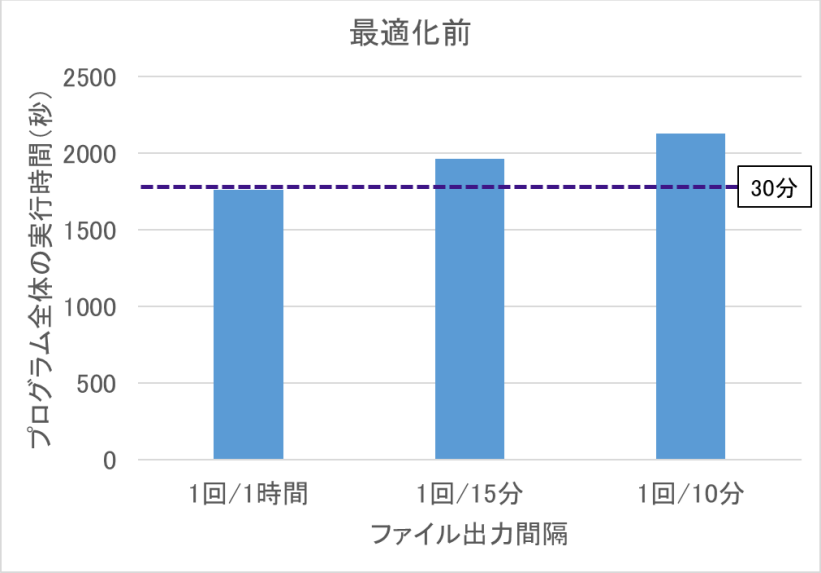

図 6.5.3-2 最適化前のプログラム全体の実行時間

(2) 最適化内容

ファイル出力処理の実行時間を削減するために，VH 上にファイル出力処理を行う MPI プロセスを立ち上げる．ファイル出力処理を行う I/O プログラムを新規に作成し（VH application），VE 上で主要演算を行うプログラム（VE application）と連携させる．図 6.5.3-3 に VE application コードを記載する．まず，VH 上の MPI プロセスと VE 上の MPI プロセスのデータのやり取りは，すべての MPI プロセスが所属するコミュニケータ（mpi_comm_world）を使用して行われる．次に，mpi_comm_split で mpi_comm_world を分割し，VE 上で動作する MPI プロセス間での通信を制御するためのコミュニケータ（mpi_comm_ve）を生成する．VE 上で主要演算を行う MPI プロセス間の通信は，この mpi_comm_ve を使用して行われる． ファイル出力に必要なパラメータ（ny，nx，t）および出力データ（data）は 1 対 1 通信である mpi_send を使用して転送している．この事例では，double precision 型の配列 data に格納された ny * nx 個のデータが，VH 上で動作するプロセスに送信される．簡単のために 1 ファイル分の出力の例を抜粋して記載しているが，実際には 4 ファイル分の転送処理が行われる．図 6.5.3-4 には VH application コードを記載する．VH applicaion では，VH，VE 上の全プロセスが所属するコミュニケータである mpi_comm_world を使用して，VE から mpi_secd により送信されたパラメータ，データを mpi_recv で受信している．データの受信が完了すると，受信したデータの出力処理を行う．VH application の終了判定は，終了フラグにより制御する．VE application では，すべての計算処理完了後，-1 にメッセージタグ 3 をつけて VH application に送信している．VH application ではメッセージタグ 3 のついたデータをパラメータ t として受信し，t の値が -1 の場合にプログラムを終了するようにしている．

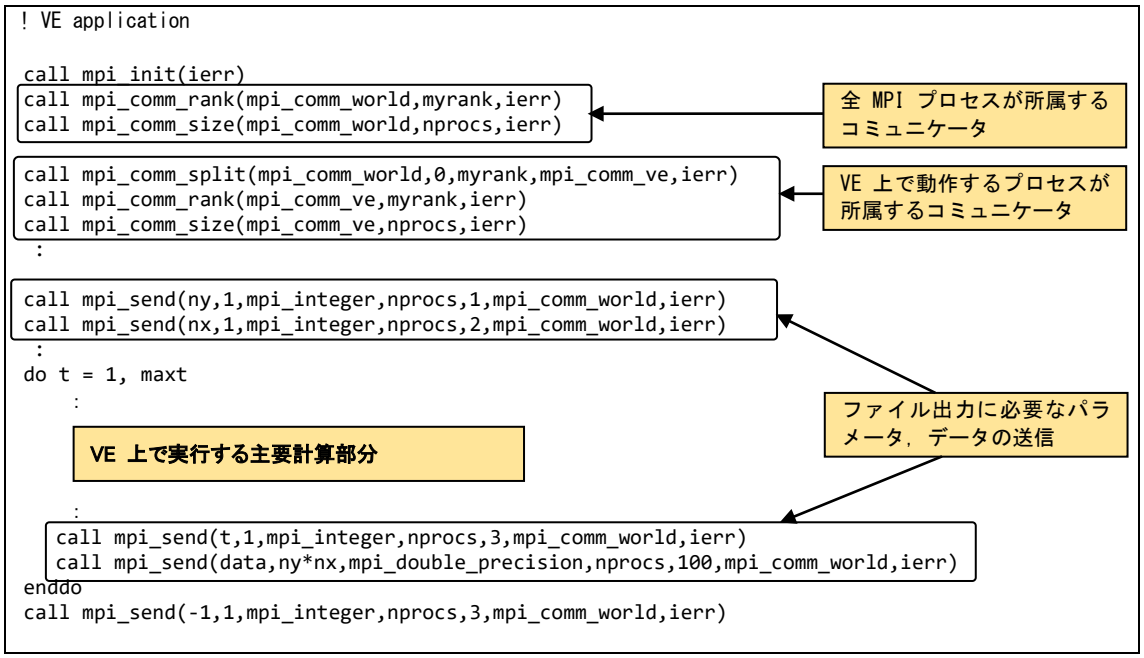

図 6.5.3-3 VE application コード

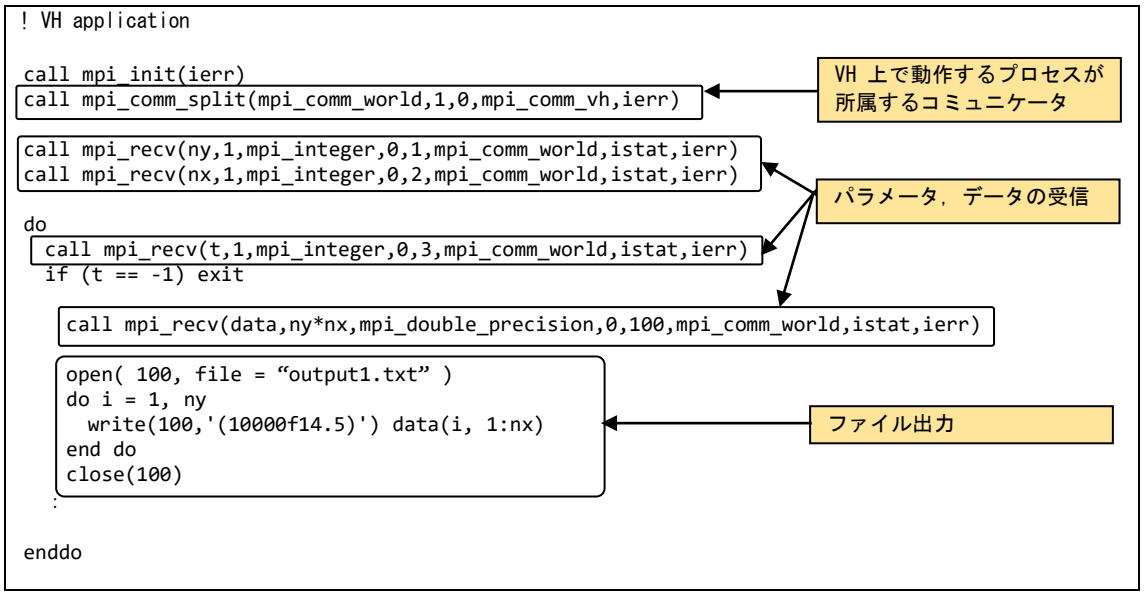

図 6.5.3-4 VH application コード

(3) 性能分析

図 6.5.3-5 に最適化前後のプログラム全体の実行時間を示す.

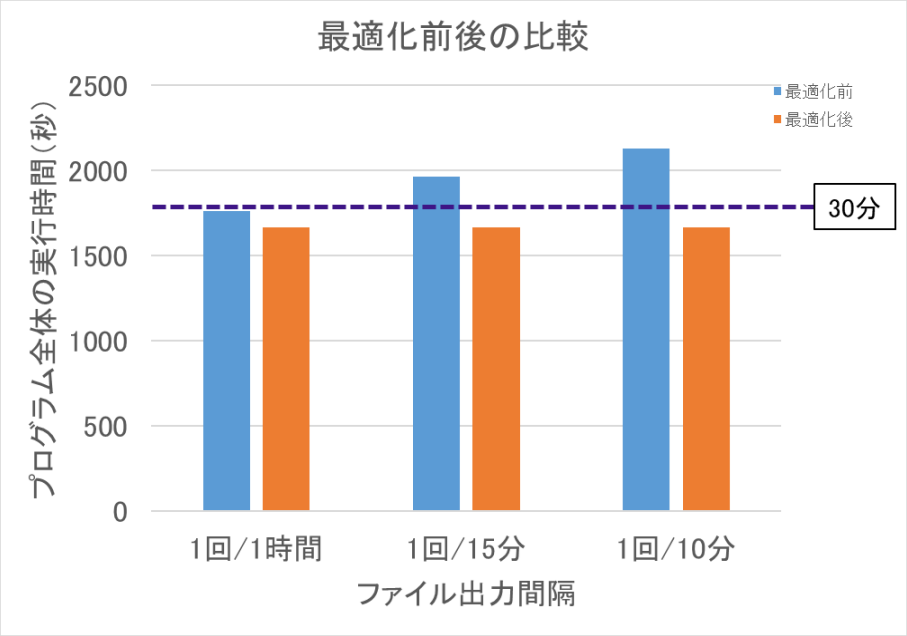

図 6.5.3-5 最適化前後のプログラム全体の実行時間

ファイル出力処理が VH 側で実行され，さらに，VE 側の演算処理と VH 側の I/O 処理がオーバーラップ可能になったことで，1 時間に 1 回出力のパターンにおいてもプログラム全体の実行時間が短縮されている．さらに，ファイル出力間隔を短くして，ファイルの出力回数を増やしても，最適化後では，プログラム全体の実行時間はほとんど変化しておらず，制限時間内にプログラムを完了させることが可能になっていることがわかる．

高速化推進研究活動報告　第 8 号

2024 年 2 月発行

編集・発行　　東北大学サイバーサイエンスセンター
〒980-8578　宮城県仙台市青葉区荒巻字青葉 6-3
https://www.cc.tohoku.ac.jp/