高速化推進研究活動報告

第7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 7 号

目 次

1. 高速化推進研究活動報告の刊行にあたって ………………………………… 1
サイバーサイエンスセンター長 曽根秀昭

2. 高速化推進研究活動報告 ……………………………………………… 2
スーパーコンピューティング研究部 江川隆輔 滝沢寛之

3. 高速化推進研究活動の成果 …………………………………………… 8
情報部情報基盤課 大泉健治 小野敏 山下毅
齋藤敦子 佐々木大輔 森谷友映

4. 大規模科学計算システムの構成 ……………………………………… 14
スーパーコンピューティング研究部 滝沢寛之 江川隆輔
情報部情報基盤課 大泉健治 小野敏 山下毅
齋藤敦子 佐々木大輔 森谷友映
高性能計算技術開発(NEC)共同研究部門 撫佐昭裕 渡部修 曽我隆
山口健太 下村陽一
NEC ソリューションイノベータ株式会社 坂口祐太

5. スーパーコンピュータ SX-ACE の高速化の概要 ……………………………… 20
スーパーコンピューティング研究部 滝沢寛之 江川隆輔
情報部情報基盤課 大泉健治 小野敏 山下毅
齋藤敦子 佐々木大輔 森谷友映
高性能計算技術開発(NEC)共同研究部門 撫佐昭裕 渡部修 曽我隆
山口健太 下村陽一
NEC ソリューションイノベータ株式会社 坂口祐太

6. 高速化事例 ………………………………………………………… 36
スーパーコンピューティング研究部 滝沢寛之 江川隆輔
情報部情報基盤課 大泉健治 小野敏 山下毅
齋藤敦子 佐々木大輔 森谷友映
高性能計算技術開発(NEC)共同研究部門 撫佐昭裕 渡部修 曽我隆
山口健太 下村陽一
NEC ソリューションイノベータ株式会社 坂口祐太

7. 三次元可視化と可視化事例の紹介 …………………………………… 87
情報部情報基盤課 大泉健治 小野敏 山下毅
齋藤敦子 佐々木大輔 森谷友映

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　曽根 秀昭

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 年に設置された大型計算機センターを起源としており，来年 2019 年は 50 周年という節目の年となります．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，設立以来およそ半世紀にわたって，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．本センターを取り巻く環境は近年大きく変化し，2007 年に最先端研究施設共用イノベーション事業により民間企業による利用の開始，2010 年に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 年に High Performance Computing Infrastructure（HPCI）への資源提供の開始等を経て，その役割が大きく広がりました．それに伴って，本センターによる利用支援および共同研究活動には，質と量の両面において格段の充実が求められてきました．その期待に応えるべく，教員はもちろんのこと，本センターの技術職員もそれらの活動の中心的な役割を果たしてきたことは，特に強調しておきたいと思います．

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．本センターでは，上記 3 者の協力体制をさらに維持・発展させるために，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．本研究部門では，本学の教員 4 名，情報基盤課の技術職員 6 名，客員教授 1 名に加えて企業から客員教授 1 名，客員准教授 1 名を迎え，さらに 2017 年からは専任教員（准教授）も任用して産学連携研究教育活動を精力的に推進しています．高性能計算技術に関するこのような取り組みは全国でも稀であり，本センターの大きな特長の一つとなっています．

今回お届けする「高速化推進研究活動報告第7号」は，以上の利用支援や共同研究等の活動の成果をまとめたもので，2015年から2018年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，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　江川隆輔　滝沢寛之

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，津波浸水被害予測，熱中症リスク啓発など，我々の安全で健康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登場など，システムの複雑化が進み，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「京」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の概要を述べる．本システムは2015年4月から本格運用されているベクトル型スーパーコンピュータSX-ACE（2,560ノード，707TFlop/s，160TB）と2014年4月に導入したスカラ型並列コンピュータLX406Re-2（68ノード，31.3TFlop/s，8.5TB）から構成されている．主力システムはその規模が示すとおりSX-ACEであり，主にユーザが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，SX-ACEには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LX406Re-2が活用され，両システムが相補的な役割を担っている．

ベクトル型スーパーコンピュータSX-ACEは2,560のノードから構成され，各ノードは4つのコア，64GBのメモリを有している．SX-ACEシステムの理論演算性能は707TFlop/s，総メモリバンド幅655TB/s，総メモリ容量160TBに達し，大規模シミュレーションを可能にしている． 図 2.2-1 に大規模科学計算システムを示す．ベクトル型スーパーコンピュータSX-ACE，スカラ型並列コンピュータLX406Re-2に加え，4PBの大規模共有ディスクと三次元可視化可能な没入型タイルドディスプレイを備えており，大規模シミュレーション結果の詳細な解析を可能としている．

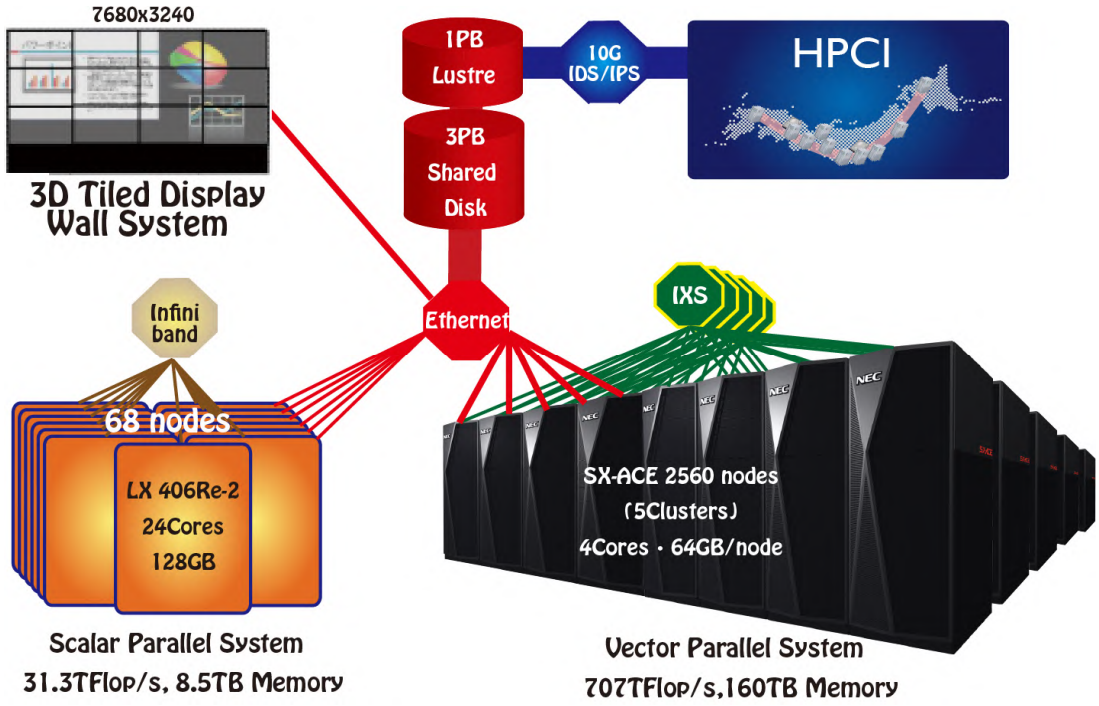


図 2.2-1 大規模科学計算システム

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，「京」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 公募研究は京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 2.3-2 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，HPCI 採択課題へとステップアップしているためであり，我々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．

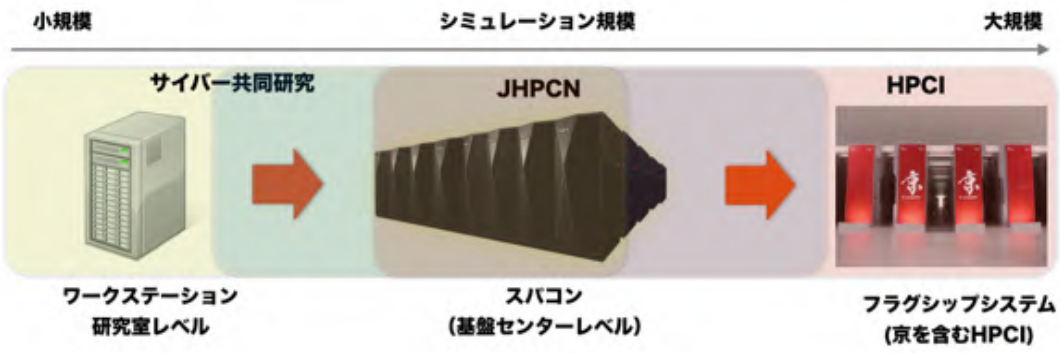


図 2.3-1 共同研究制度とシミュレーション規模

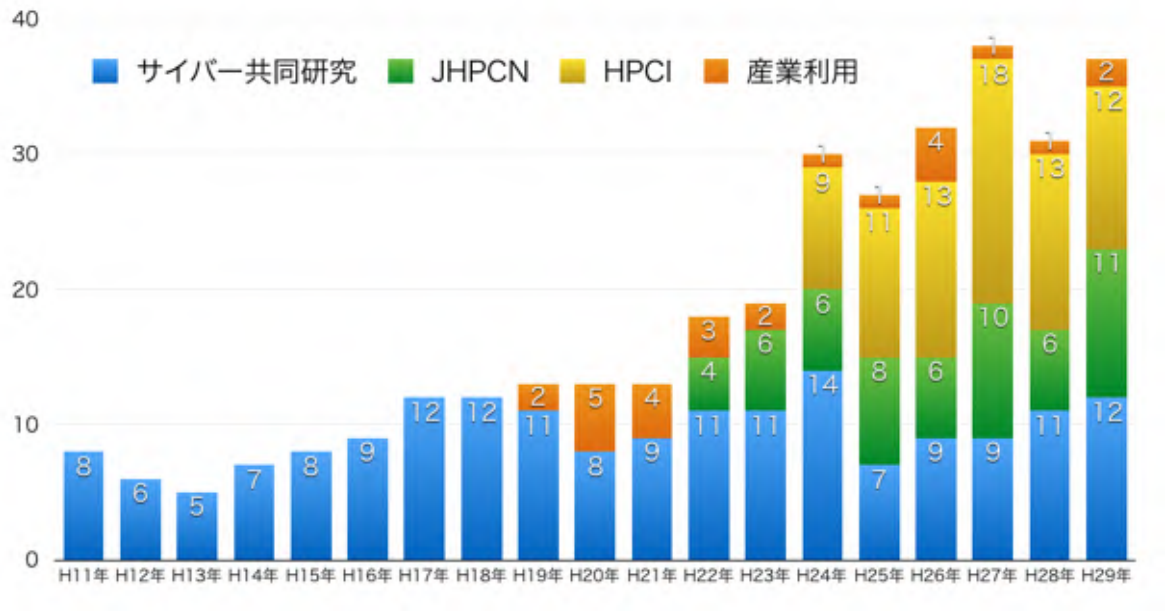


図 2.3-2 共同研究数の推移

ものづくり分野では，YS-11以来半世紀ぶりとなる国産旅客機，三菱リージョナルジェット（以下MRJ)の設計・開発に2008年から取り組んでいる．MRJの設計開発においては，東北大学で開発された非構造格子ソルバーTAS（Tohoku university Aerodynamic Simulation）コードをSX-ACEに移植・最適化を施すことで，空力に関連する設計リスクを最小限に抑制しながら，飛行試験における安全上のリスク低減に貢献している．

また，近年ではものづくりに限らず，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．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．

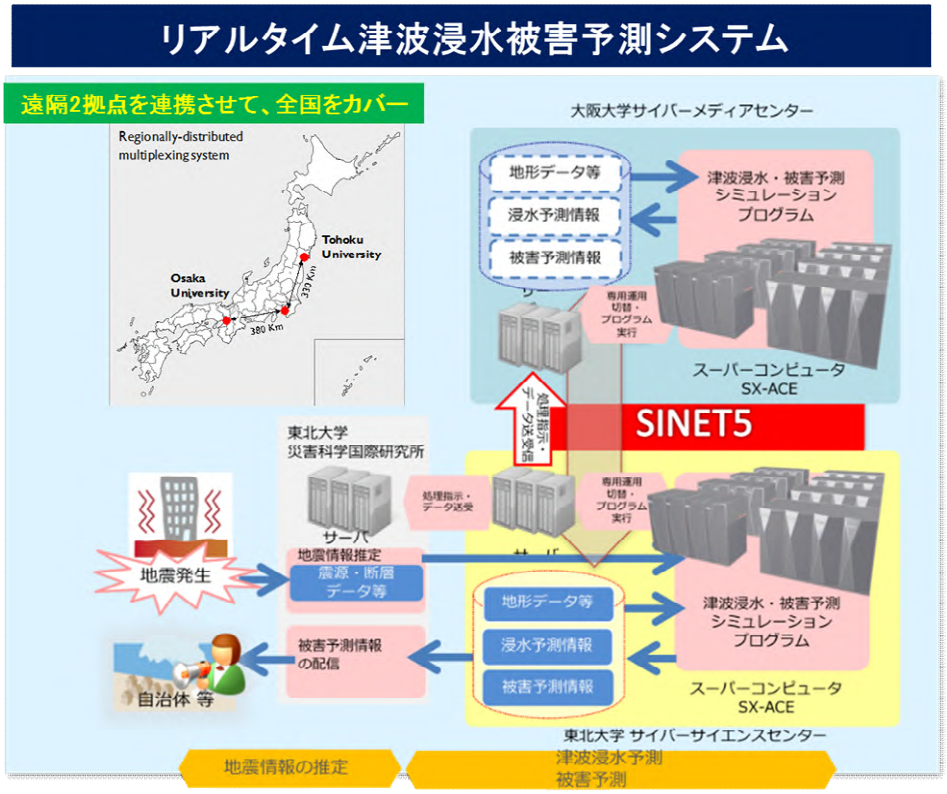


図 2.3-3 津波浸水被害予測システム

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，将来のシステムに向けた高効率プログラム開発環境と高信頼性を実現するためのチェックポイントリスタート機構，高効率運用のためのジョブスケジューリング機構の研究開発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-ACE 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．

併せて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 SC17 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2015年2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

情報部情報基盤課　大泉健治　小野敏　山下毅　齋藤敦子　佐々木大輔　森谷友映

本センターのスーパーコンピュータ SX-ACE は 2015 年 2 月にサービスを開始し，この間に利用者プログラムのシミュレーションモデルの規模はより大きくなりジョブの大規模化・長時間化の傾向は一層強くなった．本センターでは従来から，研究室では実行不可能な大規模・長時間ジョブの実行を可能とするため，スーパーコンピュータは並列処理を運用の中心に置き，実行時間の最大値は 1 ヶ月とする利用環境を整備してきた．

このような大規模・長時間ジョブを高速に処理するには，ベクトル化率と並列化率を可能な限り高めておくことが重要である．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．そこで本センターでは，利用者プログラムの高速化支援を行う担当者がベクトル最適化・MPI 並列化等の高速化支援活動を実施している．

以下では，2014 年度から2017 年度までの高速化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．なお，2014 年度については SX-9 システム向け高速化を含む．

3.1. 高速化推進研究活動（2014 年度～2017 年度）

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，表 3.1-1 に 2014 年度（平成 26 年度），表 3.1-2 に 2015 年度（平成 27 年度），表 3.1-3 に 2016 年度（平成 28 年度），表 3.1-4 に 2017 年度（平成 29 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単体性能は，シングルプログラムの高速化前後の演算時間比を示す．また並列性能は，並列プログラムの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　2014 年度（平成 26 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	
		単体性能	並列性能
1	ファイルアクセス方法の変更 インライン展開によるベクトル化の促進 作業配列の導入によるベクトル化の促進 MPI による並列化	49 倍	23 倍 (32CPU 並列)
2	指示行によるベクトル化の促進 指示行による並列性能の向上		9.4 倍 (4 コア並列)
3	インライン展開の促進による高速化	1.5 倍	

4	多重ループの一重化によるベクトル長の拡大 内側ループの手動展開によるベクトル長の拡大 ループ分割・入れ換えによるベクトル化の促進 手動インライン展開による重複演算の削除	6.5 倍	
5	MPI によるデータ転送量の削減		4.0 倍 (256CPU 並列)
6	多重ループの一重化によるベクトル長の拡大 ループ分割・入れ換えによるベクトル化の促進 自動インライン展開によるベクトル化の促進	2.7 倍	
7	コンパイルオプションの最適化	1.1 倍	
8	メモリ確保/解放タイミングの最適化 使用メモリ領域の最適化 確保したメモリ領域の再利用		1.2～2.4 倍※ (4 コア並列)
9	ループ内の IF 文の最適化	1.2 倍	1.1 倍 (256 コア並列)

※ 入力データによって性能向上比が異なる.

表 3.1-2 2015 年度(平成 27 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	
		単体性能	並列性能
1	作業配列の導入によるベクトル化の促進 ADB ヒット率の改善 未並列ループの MPI による並列化		1.8 倍 (64 コア並列)
2	MPI 分割方法の改良によるロードインバランスの改善		1.1 倍 (256 コア並列)
3	MPI 転送処理の最適化によるデータ転送量の削減		1.4 倍 (32 コア並列)
4	MPI による並列化		15.5 倍 (16 コア並列)
5	自動インライン展開によるベクトル化の促進 ループ分割によるベクトル化の促進 ファイルアクセス方法の変更	32 倍	
6	作業配列の導入によるベクトル化の促進 コンパイラ指示行によるメモリアクセス性能の改善 ファイルアクセス方法の変更	32 倍	

表 3.1-3 2016 年度(平成 28 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	
		単体性能	並列性能
1	自動インライン展開によるベクトル化の促進 ループ展開によるベクトル化の促進 ループ分割, ループ交換によるベクトル化の促進	45 倍	
2	自動インライン展開によるベクトル化の促進 ループ分割, ループ交換によるベクトル化の促進 コンパイラ指示行によるベクトル化の促進 ファイル出力方法の変更	43 倍	
3	ストリップマイニングによるメモリアクセス性能の改善	3.3 倍	
4	ASL ライブラリへの置換 ループの一重化とループ融合によるメモリアクセス性能の改善 配列サイズの変更によるメモリバンクコンフリクトの改善	18 倍	
5	MPI_ISSEND の MPI_ISEND への変更による通信性能の効率化 不必要な MPI_BARIIER の削除		1.1～1.5 倍 (32 コア並列)
6	作業配列の導入によるベクトル化の促進	3 倍	
7	リダクション処理(ALLREDUCE)の最適化 ファイル出力方法の変更		3.8 倍 (636 コア並列)
8	多重ループの融合/分割/入れ換えによるベクトル化の促進 IF 文のループ外への移動によるベクトル化の促進 MAX・MIN 関数への置き換え, 除算の乗算化, 冗長演算の削除による演算の効率化 作業配列の変数化によるメモリアクセス性能の改善 RedBlack 法の間接参照からマスク処理への変更によるメモリアクセス性能の改善 MPI による並列化	3 倍	5.1 倍 (4→16 コア並列)
9	GTHREORDER 指示行の挿入によるリストベクトルアクセスの効率化 MPI_ISSEND の MPI_ISEND への変更による通信性能の効率化		1.3～1.4 倍 (32 コア並列)

表 3.1-4 2017 年度(平成 29 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	
		単体性能	並列性能
1	グローバルメモリ機能の使用による通信性能の改善 非同期通信への変更による通信性能の改善 演算オーバーラップ機能の使用		1.2 倍 (32 コア並列)
2	配列定義の変更による平均ベクトル長の改善 指示行による再内ループの展開 指示行によるメモリアクセス性能の改善	1.8 倍	
3	指示行の挿入によるベクトル化の促進 ループブロック化, マスク処理, ループ交換によるベクトル化の促進	10 倍	
4	計算カーネル部分について, ループブロック化, マスク処理によるベクトル化の促進	1.7 倍	
5	不要な演算の削減 依存関係解消のための作業配列追加によるベクトル化の促進 MPI 通信性能の改善 ハイブリッド並列の効率化 メモリ使用量の削減		約 5000 倍 (512 コア並列・推定値)
6	指示行の挿入による演算効率の改善 指示行の挿入によるメモリアクセス性能の改善		1.3 倍 (32 コア並列)
7	複雑な条件分岐の簡略化のためのループ分割によるベクトル化の促進 通信命令の並び替えによる通信性能の改善	10 倍	1.05 倍 (636 コア並列)

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

SX-ACE システムの 2014 年度から 2017 年度までの各年度における, ジョブのベクトル化率および並列化率と, そのような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. SX-ACE を導入した 2014 年度において特徴的なのは, 並列化率は 80% 以上であるがベクトル化率がほぼ 0% であるジョブのノード時間割合が, 15% 程度あった点である. これは他のシステムにおいて高度に並列化されたジョブがSX-ACEで実行されたため, 十分にベクトル化されていなかったと考えられる. 高速化支援によってこのようなジョブのベクトル化が促進されたことにより, 翌年の 2015 年度ではベクトル化率および並列化率ともに 80% を超えるジョブのノード時間割合は 50% を上回った. また, 2016 年度においては新規利用者と思われるベクトル化率および並列化率ともに 10% 未満のジョブが見られたが, 高速化支援によって 2017 年度ではベクトル化率についてはほぼ 80% 以上となっている.

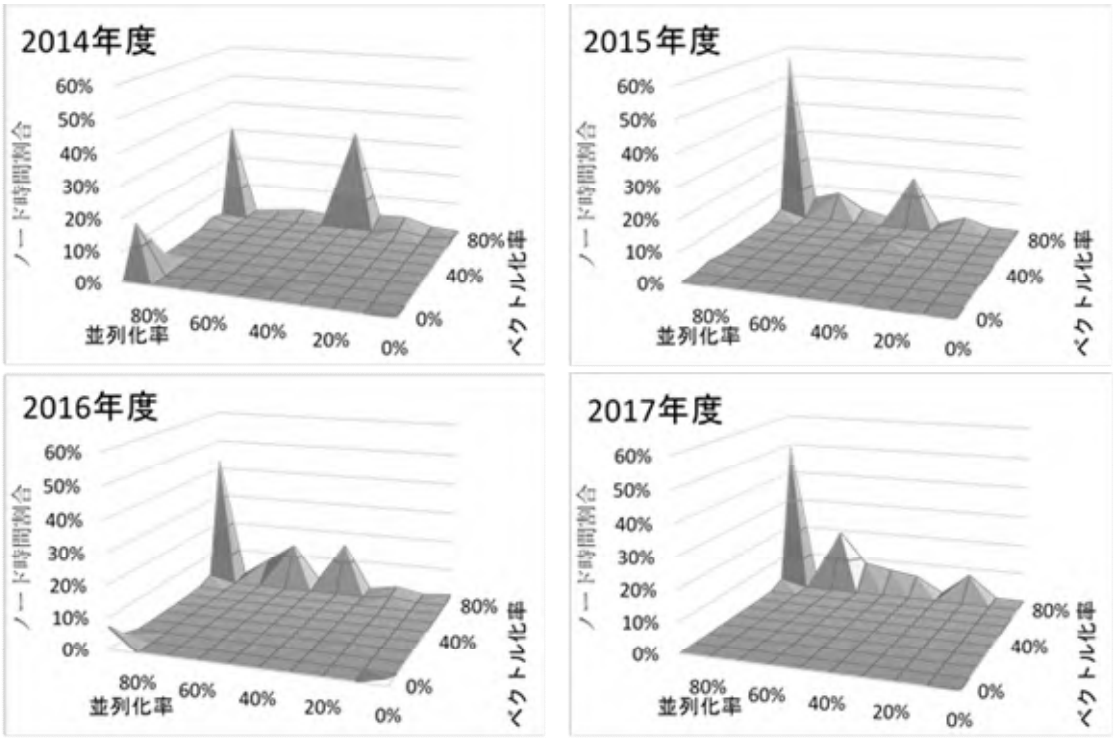


図 3.2-1 ベクトル化率および並列化率とノード時間割合

4 年間の利用者ジョブのベクトル化率と並列化率の状況を表 3.2-1 示す. 表 3.2-1 は SX-ACE で実行した利用者ジョブをベクトル化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 からベクトル化率と並列化率に関して 3 つのカテゴリで分類したものが表 3.2-2 である.

表 3.2-1 ベクトル化率と並列化率の状況

ベクトル化率 \ 並列化率	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
90%	0	0	4	0	14	3	8	12	2	48
80%	0	0	1	0	0	0	0	0	0	2
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	1
10%	0	0	0	0	0	0	0	0	0	2
0%	1	0	0	0	0	0	0	0	0	2

表 3.2-2 ベクトル化率と並列化率の状況

ベクトル化率と並列化率の分類	ノード時間の割合
ベクトル化率と並列化率の双方が 90% 以上のジョブ	約 48%
ベクトル化率が 90% 以上のジョブ	約 91%
並列化率が 90% 以上のジョブ	約 55%

表 3.2-2 のように，ベクトル化率，並列化率がともに高く，SX-ACE の特性が十分に活かされているジョブのノード時間の割合が半分程度を占める状況は，高速化推進研究活動の成果であると考えられる．

表 3.2-3 は 2014 年度から 2017 年度までの 4 年間について，ジョブのノード時間ごとのジョブの数の割合を示したものである．2014 年度および 2015 年度においては 100,000 ノード時間以上のジョブの割合が 10%未満であったが，2017 年度においては 34% とジョブの大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	2014 年度	2015 年度	2016 年度	2017 年度
0 ～ 999	13%	20%	17%	20%
1,000 ～ 9,999	25%	40%	28%	19%
10,000 ～ 49,999	32%	20%	11%	17%
50,000 ～ 99,999	22%	16%	11%	10%
100,000 ～	8%	4%	33%	34%

3.3. 今後の取り組み

3.2節に述べたようにSX-ACEの運用期間中，ジョブの大規模化・長時間化が進んだ．ただし，ベクトル化率，並列化率とも高いところに集中しているが，並列化率90% 未満のジョブが実行される割合も45% あった．このことより，SX-ACE向けの高速化チューニングはもとより，MPI化を含む並列処理のチューニングに力を入れた高速化支援を今後も行っていく必要があると考えている．

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　江川隆輔
情報部情報基盤課　大泉健治　小野敏　山下毅　齋藤敦子　佐々木大輔　森谷友映
高性能計算技術開発(NEC)共同研究部門　撫佐昭裕　渡部修　曽我隆　山口健太　下村陽一
NEC ソリューションイノベータ株式会社　坂口祐太

本センターは，全国共同利用施設として，スーパーコンピュータシステム SX-ACE，並列コンピュータシステム LX406Re-2，ファイルサーバシステム，三次元可視化システムの提供を行っている．図 4-1 に，本センターのシステム構成を示す．

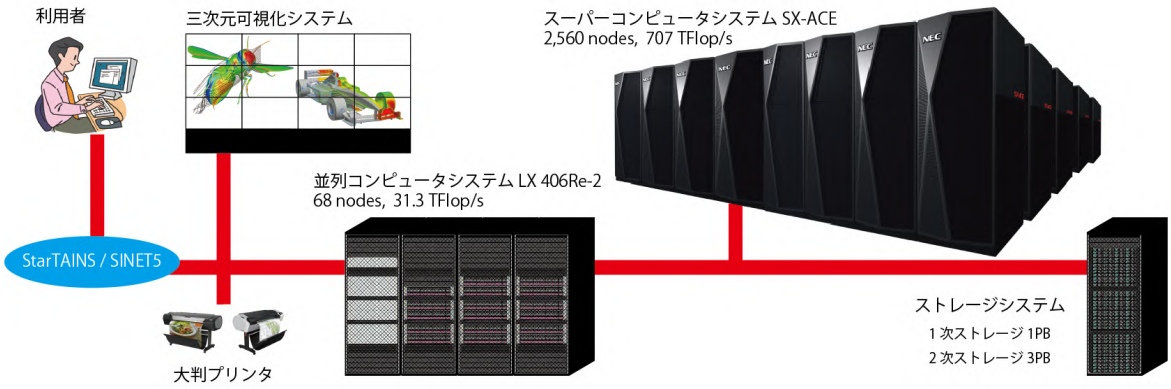


図 4-1 システム構成

スーパーコンピュータシステム SX-ACE および並列コンピュータシステム LX406Re-2 の特徴については，それぞれ「4.1. SX-ACE の特徴」，「4.2. LX406Re-2 の特徴」に記載する．

ファイルサーバシステムは，1PB の一次ストレージ領域と，3PB の二次ストレージ領域を提供している．

一次ストレージは，DDN 社製の Lustre ファイルシステムと Gfarm ファイルシステムで構成される．一次ストレージは並列コンピュータシステムと 6.8GB/s の InfiniBand で接続されており，並列コンピュータで大規模な入出力を必要とする利用者に向けて大規模ファイル領域として提供されている．

二次ストレージは，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，本センター利用者のホームディレクトリ環境として提供されている．ScaTeFS は NEC 独自のプロトコルによる高効率のデータ転送方式が用いられており，高速なファイルシステム共有が可能である．スーパーコンピュータシステム SX-ACE は，最大 40Gbps の転送性能を持つ Juniper 社製 QFabric システムを介して二次ストレージと接続されており，高速な入出力が可能である．

三次元可視化システムは，3D 対応 50 インチ LED モニタを 12 面配置した大画面タイルドディスプレイと，演算結果の可視化処理およびディスプレイへの描画を行う可視化サーバ 4 ノードで構成される．可視化アプリケーションとして AVS/Express MPE を備えており，本センターの計算機で得られたデータの三次

元可視化が可能である. また, 大画面タイルドディスプレイは, テレビ会議システムとしても利用可能である. 三次元可視化事例の紹介を「7. 三次元可視化と可視化事例の紹介」に記載する.

4.1. SX-ACE の特徴

SX-ACEの単一ノードは, 4つのコアで構成された単一CPUノードとなっている. 本センターでは2,560ノードの SX-ACE を運用しており, 総論理演算性能は 707TFlop/s(総ベクトル演算性能 655TFlop/s), 総メモリ容量は 160TB となっている.

単一ノードを多数接続するマルチノードシステムとして, 512 ノードを 1 つのクラスタとして構成した基本クラスタが 3 クラスタ, 1,024 ノードを 1 つのクラスタとして構成した拡張クラスタが 1 クラスタ利用可能であり, 利用者は, 最大 4,096 のコアを利用して計算を行うことができる.

システムの主な諸元を表 4.1-1 に示す.

表 4.1-1 SX-ACE の主要諸元

CPU ノード	CPU 数	1
	コア数	4
	理論演算性能※1／ベクトル演算性能※2	276GFlop/s ／ 256GFlop/s
	メモリ容量／メモリ帯域	64GB ／ 256GB/s
	ノード間通信性能	4GB/s × 2 (双方向)
基本クラスタ	CPU 数	512
	コア数	2,048
	理論演算性能※1／ベクトル演算性能※2	141TFlop/s ／ 131TFlop/s
	メモリ容量／メモリ帯域	32TB ／ 131TB/s
	ノード間通信性能(基本クラスタ全体)	2TB/s × 2 (双方向)
拡張クラスタ	CPU 数	1,024
	コア数	4,096
	理論演算性能※1／ベクトル演算性能※2	282TFlop/s ／ 262TFlop/s
	メモリ容量／メモリ帯域	64TB ／ 262TB/s
	ノード間通信性能(拡張クラスタ全体)	4TB/s × 2 (双方向)
システム	CPU 数	2,560
	コア数	10,240
	理論演算性能※1／ベクトル演算性能※2	707TFlop/s ／ 655TFlop/s
	メモリ容量／メモリ帯域	160TB ／ 655TB/s
	ノード間通信性能(システム全体)	10TB/s × 2 (双方向)

※1) 同時に並列演算することができるすべての演算器(ベクトルユニットの乗算演算器, 加算演算器, 除算/平方根演算器, ならびにスカラユニットの浮動小数点演算器)における浮動小数点演算処理能力の総和.

※2) 同時に並列演算することができるベクトルユニット内の乗算演算器, 加算演算器のみによる浮動小数点演算処理能力の総和.

4.1.1. ノード構成

SX-ACE の CPU は 4 つのコア，16 個のメモリ制御部（Memory Control Unit：MCU），RCU，IO 制御部(Input Output Control unit：IOC) と，クロスバー・スイッチ部から構成される．この CPU と 16 枚の DIMM(メモリ)をカード上に実装したものをノードと呼ぶ．図 4.1.1-1 に，ノード構成を記載する．

クロスバー・スイッチは，4 つのコアが同時にメモリアクセスした場合は，各コアのメモリバンド幅は平均 64GB/s となるが，1 つのコアのみを利用した場合や各コアのメモリアクセスタイミングが重ならない場合は，1 コアで 256GB/s の全メモリバンド幅を使用することができる．

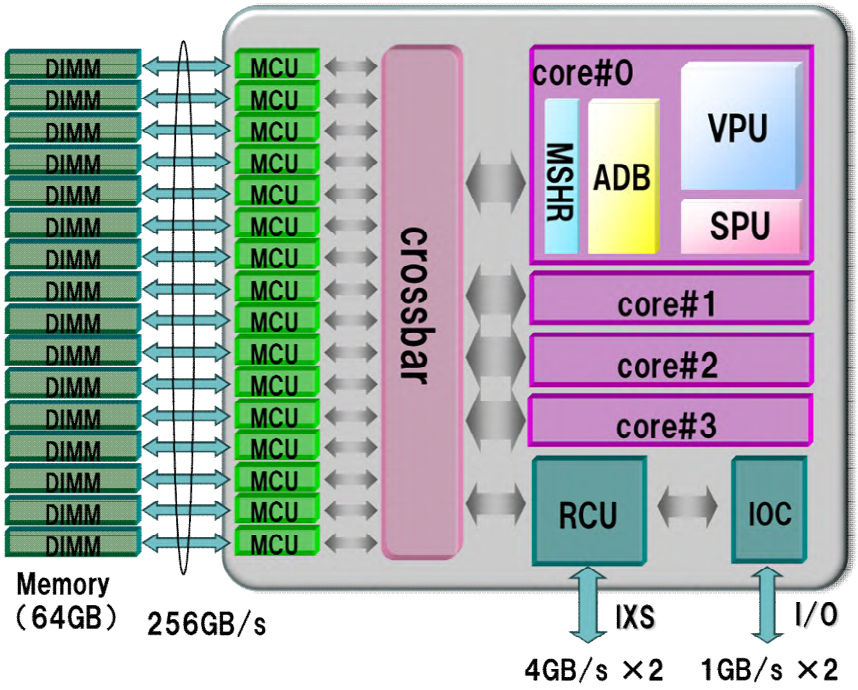


図 4.1.1-1 SX-ACE のノード構成図

図 4.1.1-2 に，SX-ACE のコア構成を示す．コアはスカラ処理部（Scalar Processing Unit：SPU），ベクトル処理部（Vector Processing Unit：VPU），ソフトウェア制御可能な高速バッファ部（Assignable Data Buffer：ADB），冗長なメモリアクセスを抑止する MSHR（Miss Status Handling Register）で構成される．SPU は全ての命令発行制御を行い，ベクトル命令やノード間通信命令を VPU やノード間通信制御部（Remote access Control Unit：RCU）に発行する．SX-ACE では，SPU から VPU への命令発行専用のパスを設け，従来の SX システムに比べて，ベクトル命令の発行をより効率よく行えるようにしている．VPU は 16 本のベクトル・パイプラインから構成され，1 つのベクトル演算命令を 1 サイクルあたり 16 演算のスループットで処理することで，コアのベクトル演算性能（ベクトル乗算・加算演算性能）は 64GFlop/s を実現している．SX-ACE で高い演算性能を実現するためには，プログラムのベクトル化率を高める，平均ベクトル長を長くするなどの最適化により，VPU を効率よく使用することが重要である．SX-ACE の最適化の事例については「6. 高速化事例」に記載する．各ベクトル・パイプラインの内部構造は，乗算器×2，加算／シフト演算器×2，除算／平方根演算器×1，論理演算器×1 の 6 種のそれぞれ独立に動作可能な演算器と，マスク演算パイプライン，ロード／ストアパイプライン，マスクレジスタ，

およびベクトルレジスタにより構成される.

SX-9 から導入された ADB は, SX-ACE ではその容量が SX-9 の 4 倍の 1MB となっている. ADB はコアと主記憶装置間に配置され, ベクトルデータのバッファリングを行う. ADB はメモリバンド幅が広く, メモリレイテンシが短いという特徴がある. SX-ACE では, ADB から VPU へは, 最大 256GB/s のデータ供給性能を備えている.

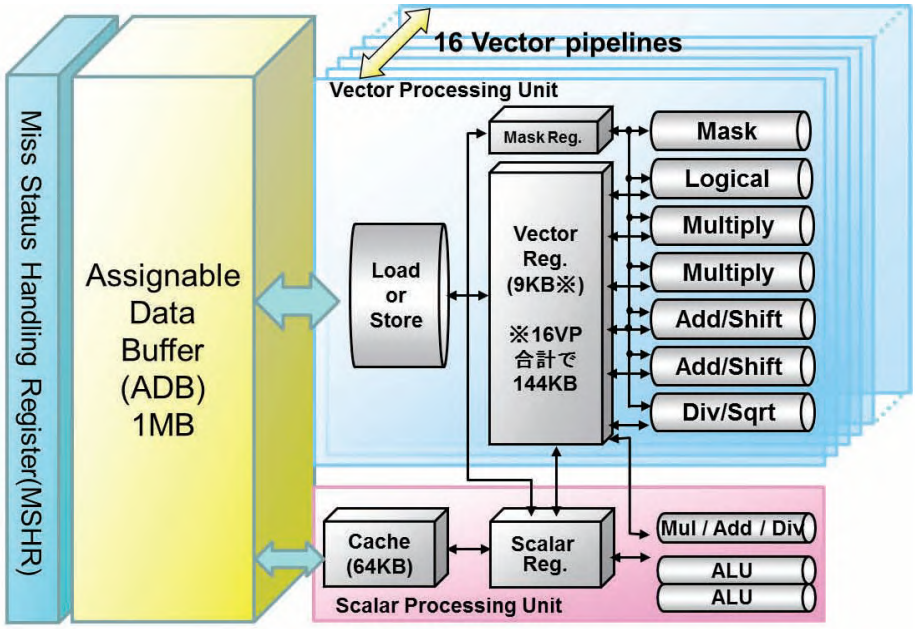


図 4.1.1-2　SX-ACE のコア構成図

4.1.2.　マルチノード構成

SX-ACE 基本クラスタのマルチノード構成を図 4.1.2-1 に示す. 基本クラスタの総理論演算性能は 141TFlop/s , 総主記憶容量は 32TB となる. ひとつの基本クラスタは 512 ノードで構成され, 各ノードは IXS を介して接続される. IXS は専用開発の 32 ポートを持つスイッチ LSI を使い, 2 段のフル・ファットツリー・トポロジーを採用している. RCU のデータ受信部, および送信部は, それぞれ独立に動作可能であり, ノードあたり 4GB/s×2(双方向)の通信バンド幅を実現している. また, RCU は CPU とは独立に動作するデータムーバ(DMA エンジン)を持つことにより, ノード間のデータ転送を CPU 動作とは独立して行うことが可能である. 基本クラスタのバイセクションバンド幅は, 2TB/s となる.

本センターでは, 512 ノードで構成された基本クラスタに加えて, 1,024 ノードで構成された拡張クラスタの運用も行っている. 拡張クラスタでは, 最大 4,096 のコア, 64TB のメモリ容量を利用可能である.

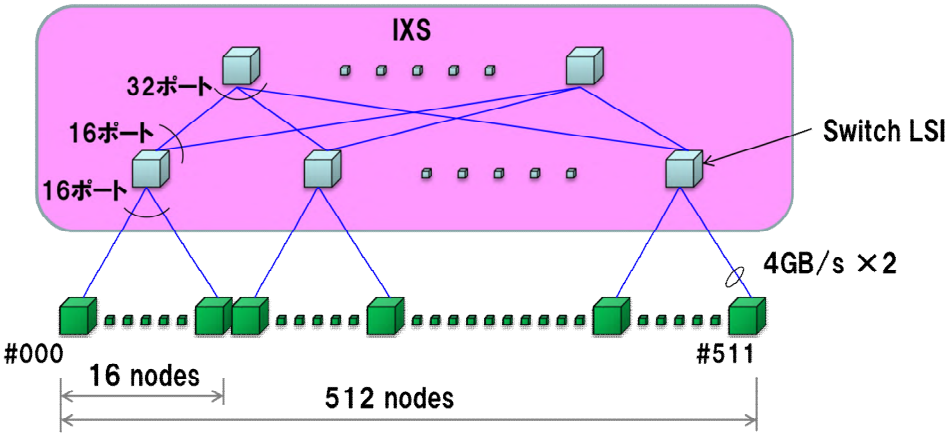


図 4.1.2-1 SX-ACE のマルチノード構成図

4.2. LX406Re-2 の特徴

LX406Re-2 は，1 ノードにインテル®Xeon®プロセッサ E5-2695v2（12 コア）を 2 基と，128GB の主記憶を搭載している．本センターでは，68 ノードの LX406Re-2 を運用しており，総理論演算性能 31.3TFlop/s（倍精度），総メモリ容量は 8.5TB を有している．

LX406Re-2 はアプリケーションサーバとしての役割も担っており，高速ディスクアクセスが可能な SSD ドライブを搭載する専用ノードにより，Gaussian 等のアプリケーションプログラムを高速に実行することができる．システムの主な諸元を表 4.2-1 に示す．

表 4.2-1 LX406Re-2 の主要諸元

ノード	CPU 数	2
	コア数	24
	理論演算性能	460.8GFlop/s
	メモリ容量	128GB
	ノード間通信性能	6.8GB/s
システム	CPU 数	136
	コア数	1,632
	理論演算性能	31.3TFlop/s
	メモリ容量	8.5TB
	ノード間通信性能（システム全体）	462.4GB/s

4.2.1. ノード構成

図 4.2.1-1 に，LX406Re-2 のノード構成を示す．LX 406Re-2 は，2 つの CPU と 8 つの DIMM で構成されたノードを基本単位とし，各ノードを高速な InfiniBand（IB）ネットワークで結合したクラスタを構築することで，大規模並列計算を行う．インテル®Xeon®プロセッサ E5-2695v2 は，CPU あたり 12 のコアを搭載しており，ノードあたり 24 コアの構成となる．256-bit SIMD 演算（乗算）および 256-bit SIMD 演算（加算）を行うことで，1 クロックサイクルあたり最大 8 回の倍精度演算を行うことができる．DDR3-1866 対応 DIMM スロットを 8 基有し，16GB DIMM を使用することで 128GB のメモリ容量を実装している．キ

ャッシュメモリは，全コア共有の L3 キャッシュが 30MB，コアごとの L2 キャッシュが 256KB，L1 キャッシュは命令，データそれぞれでコアごとに 32KB である．

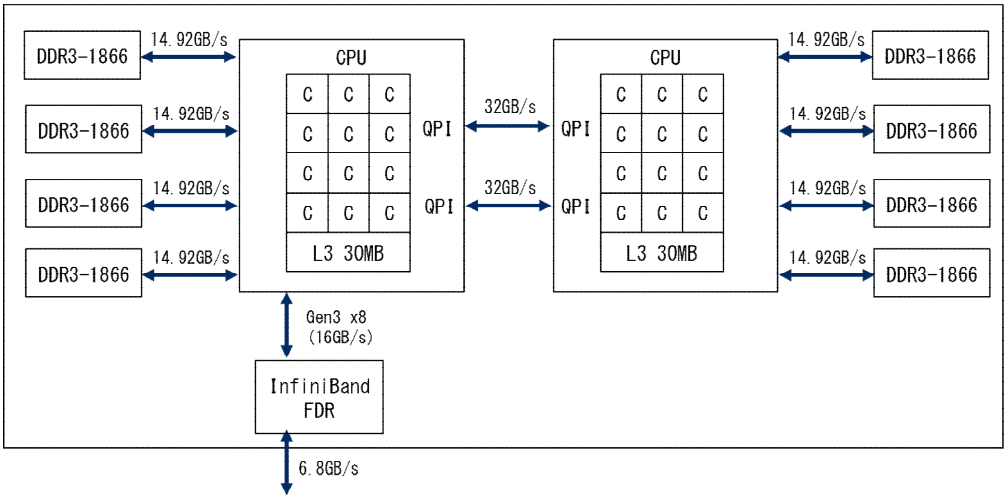


図 4.2.1-1　LX406Re-2 のノード構成図

4.2.2.　マルチノード構成

LX406Re-2 のマルチノード構成を図 4.2.2-1 に示す．68 ノードがポートあたり 6.8GB/s の転送性能を持つ IB スイッチに接続されており，フルバイセクションバンド幅の構成となっている．LX406Re-2 は 1 ノードあたり 24 コア使用可能であるため，システム全体としてのコア数は 1,632 コアとなる．本センターの運用においては，1 ジョブで利用可能なノード数は 24 ノードまでとしているため，最大で 576 プロセス，1.5TB のメモリ容量を使用した MPI プログラムを実行することができる．

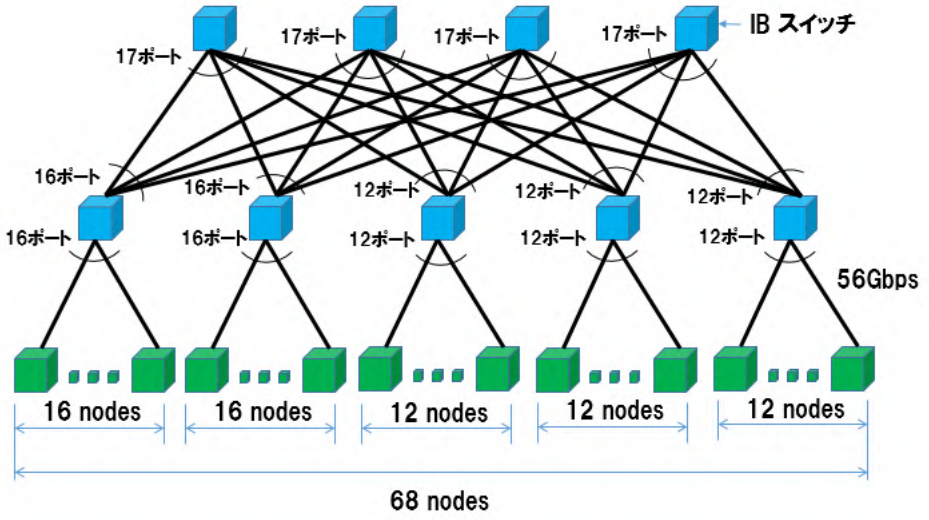


図 4.2.2-1　LX406Re-2 のマルチノード構成図

5. スーパーコンピュータ SX-ACE 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　江川隆輔
情報部情報基盤課　大泉健治　小野敏　山下毅　齋藤敦子　佐々木大輔　森谷友映
高性能計算技術開発(NEC)共同研究部門　撫佐昭裕　渡部修　曽我隆　山口健太　下村陽一
NEC ソリューションイノベータ株式会社　坂口祐太

5.1. ベクトル処理

5.1.1. ベクトル処理の概要

科学技術計算プログラムを実行する場合，特定のループに実行コストの大部分が集中することが多い．ループ内で繰り返される，規則的に並んだ配列データ(ベクトルデータ)の演算をベクトルユニット(VPU)で一括実行することをベクトル処理と呼ぶ．SX-ACE でプログラムを高速に実行するためには，VPU を効率的に動作させる必要がある．

一方で，スカラ処理は，ループ内の繰り返し演算を逐次的に実行するものである．

5.1.2. ベクトル化率

ベクトル化率とは，あるプログラムをスカラ処理で実行する場合の総実行時間に対して，そのプログラムでベクトル処理が可能な部分の実行時間の割合を表す．図 5.1.2-1 にプログラムを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処理で実行した場合の実行時間の概念図を示す．

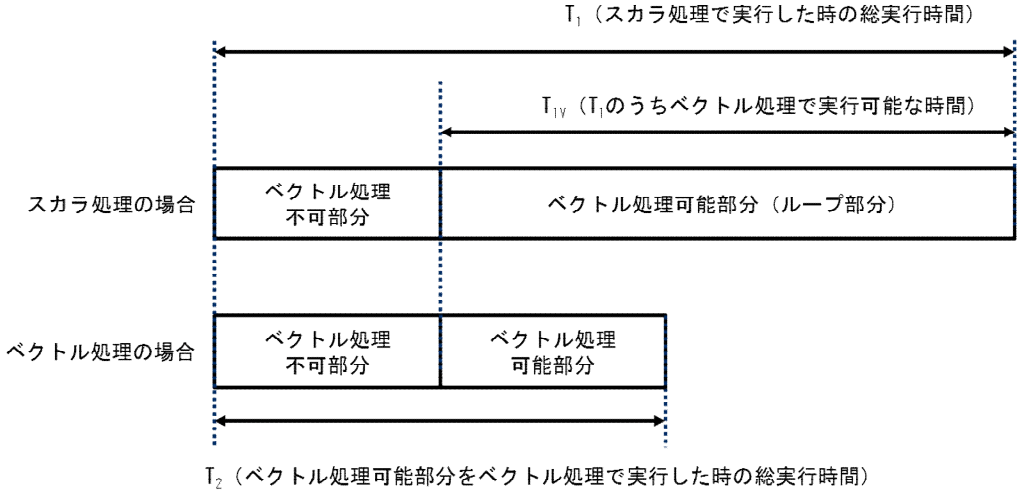


図 5.1.2-1 プログラムの実行時間の概念図

プログラムには，ベクトル処理不可の部分とベクトル処理可能な部分があり，ベクトル処理可能な部分はベクトルユニットで一括処理されることで実行時間が短縮される．プログラムをスカラ処理で実行する場合の総実行時間を T_1，ベクトル処理が可能な部分の実行時間を T_{1V} とすると，ベクトル化率 α は

以下の式で定義される.

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

ベクトル化率を求めるためには，プログラムをすべてスカラ処理で実行した場合の実行時間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. しかし，これらの時間は同時には得られないためベクトル化率を容易に算出することは困難である. そこで，SX-ACE ではベクトル処理が十分に行われているかの確認指標として，ベクトル化率の代わりにベクトル演算率を使用する. 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定義され，ほぼベクトル化率とみなせる. ベクトル演算率は，後述の性能解析ツールを使用することで出力されるため，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.

そのプログラムをベクトル処理する場合の性能向上比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表される.

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

この式からベクトル化率と性能向上比の関係は図 5.1.2-2 のようになり，この関係をアムダールの法則とよぶ. アムダールの法則から高い性能向上を実現するためには，ベクトル化率を 100% に可能な限り近づけることが必要であることがわかる.

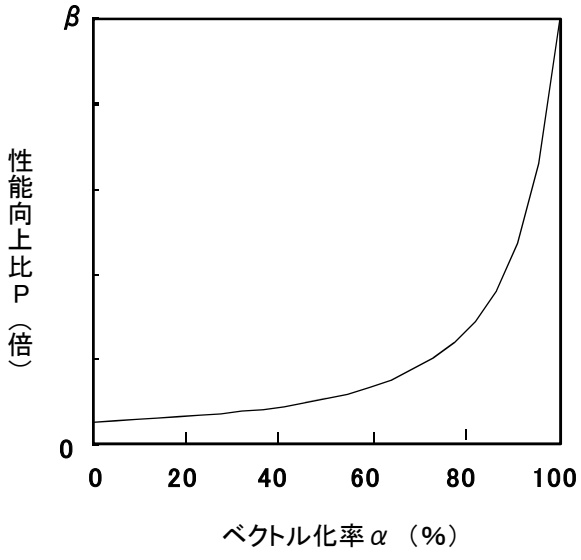


図 5.1.2-2　ベクトル化率と性能向上比の関係（アムダールの法則）

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をベクトル長(ループ長)と呼ぶ. SX-ACE では, 1 回のベクトル命令で 256 要素のベクトル処理を行うことができ, 1 回のベクトル命令で演算を行った演算要素数の平均値を平均ベクトル長と呼ぶ. 図 5.1.3-1 に記載したループの場合, ループ長 300 のループは, 256 要素の演算と 44 要素の演算の 2 回で実行される. この場合の平均ベクトル長は 150 (=(256+44)/2) となる. ベクトル処理を効率よく行うためには, ループ長を長くし, 平均ベクトル長を最大値である 256 に近づけるようにする必要がある.

```
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に少し時間がかかる. これを立ち上がり時間と呼ぶ. そのため, ループ長が非常に短い場合, ベクトル化するよりもベクトル化しないで処理したほうが実行時間は短くなる. 図 5.1.3-2 に, プログラムをベクトル化した場合とベクトル化しない場合について, 実行時間とループ長の関係を示す.

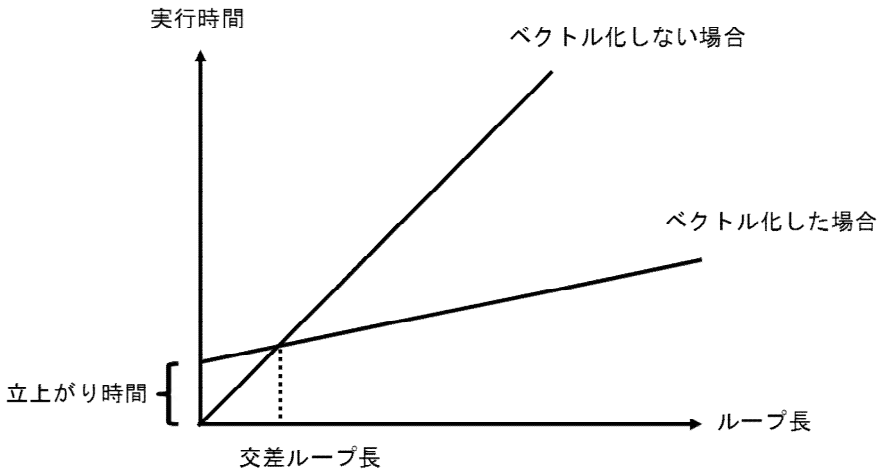


図 5.1.3-2 交差ループ長

ベクトル化した場合とベクトル化しない場合で実行時間が同程度となるベクトル長を交差ループ長と呼ぶ. SX-ACE の場合, 交差ループ長は5程度である. ベクトル長をできるだけ長くすることで, 高速化の効果が大きくなることがわかる.

5.2. メモリアクセス処理

5.2.1. メモリアクセス

ベクトル処理を高速に行うためには，主記憶装置からベクトルユニットに対して，演算に必要なデータを滞りなく提供することが必要である．

データ供給の遅延の要因にバンクコンフリクトがある．SX-ACE のメモリは高いバンド幅を実現するため 128 個のメモリバンクから構成される．SX-ACE はコアあたり 16 個の CPU ポートをもち，メモリバンクと接続されている．図 5.2.1-1 に，SX-ACE のコアとメモリバンクの接続概念図を示す．

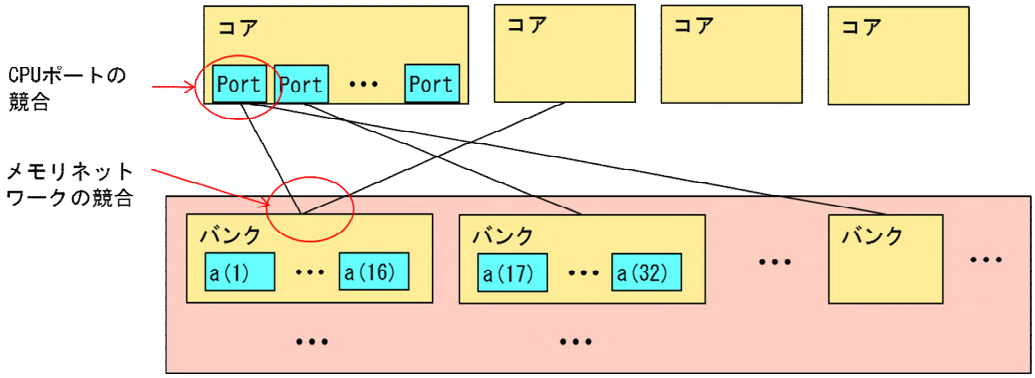


図 5.2.1-1 コアとメモリバンクの接続概念図

バンクコンフリクトは CPU 競合とメモリネットワーク競合の 2 つに分類される．CPU ポート競合は CPU 内における同一ポートにロード・ストアが集中した場合に発生する競合を指す．メモリネットワーク競合は，同一のメモリバンクへのアクセスで発生する競合や CPU と主記憶装置間の経路上で発生する競合を指す．バンクコンフリクトが発生すると，CPU へのデータ供給能力が低下し，高速化が阻害される要因となる．バンクコンフリクトを回避するための対処として，メモリアクセスが連続アクセスとなるようなプログラム修正やループアンローリングなどによるメモリアクセス回数の削減などが必要である．

5.2.2. キャッシュアクセス

SX-ACE では，ADB（Assignable Data Buffer）と呼ばれるベクトルデータ・バッファリング機構を利用することで，ベクトル命令によるメモリアクセスを改善することができる．図 5.2.2-1 に ADB を記載する．

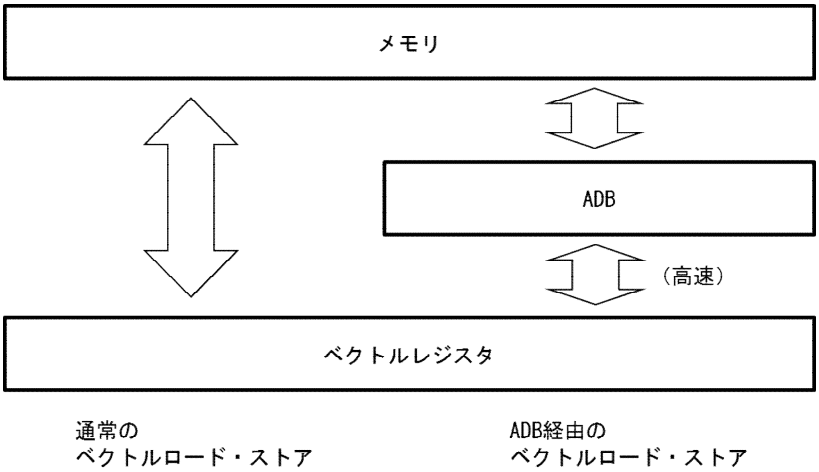


図 5.2.2-1 ADB

メモリにアクセスする通常のベクトルロード・ストアに比べて，ADB 経由のベクトルロード・ストアは高速に処理される．ADB を利用する場合，配列データの最初のベクトルロード・ストア時にデータが ADB にバッファリングされる．次にそのデータを利用するときには，ADB にバッファリングされたデータがロードされるため，2 回目以降のベクトルロードを高速化できる．

5.3. 並列処理

5.3.1. 並列化の概要

並列処理とは，プログラムの中で並列に処理可能な部分を複数のコアを使用して実行することである．並列処理を可能とするために処理の分割を行うことを並列化と呼ぶ．並列化を行うことでプログラムの実行時間（経過時間）を短縮することができる．

並列処理にはシステムのアーキテクチャに応じた以下の方式がある．

5.3.2. 共有メモリ並列

共有メモリ並列とは，図 5.3.2-1 のように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．利用可能な並列化手法として自動並列や OpenMP 並列がある．

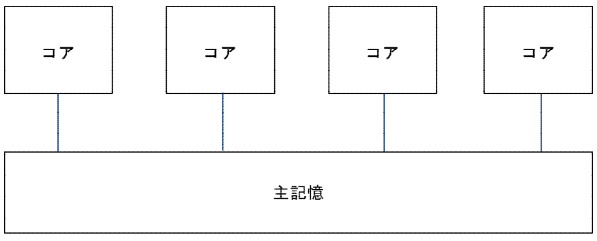


図 5.3.2-1 共有メモリ並列

(1) 自動並列

SX-ACE は，1 ノードあたり 64GB のメモリを搭載しており，自動並列化機能を使用することで 4 個のコアでメモリを共有することができる．一般的に並列化を行う場合には，並列化しても結果が変わ

らないことを保証するためにデータの依存関係の解析を行い，プログラムの変形や指示行の挿入を行う必要がある．しかし，SX-ACE の自動並列機能を利用すると，それらの作業をコンパイラが自動的に行う．

SX-ACE のコンパイラでは，コンパイルオプション「-Pauto」を指定してコンパイルするだけで自動並列機能を使用することができる．コンパイラは，依存関係の解析，タスクの生成，データの分割などを全て自動で行う．

(2) OpenMP 並列

自動並列以外の共有メモリ並列の実装方法として OpenMP 並列がある．コンパイラが自動的に並列化を行う自動並列に対し，OpenMP による並列化は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．コンパイル時に OpenMP を使用するためのオプションを指定することで並列化が行われる．SX-ACE のコンパイラでは，「-Popenmp」を指定することで OpenMP 指示行が有効となる．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そのままの形でコンパイル，実行することができ，ポータビリティ(portability，可搬性)に優れている．

Fortran プログラムの場合，OpenMP 指示行は「!$OMP [指示文]」を 1 カラム目から指定する．DO ループを並列化する場合は，DO ループの前に「parallel do」を指定し，ENDDO 文の後ろに「end parallel do」を指定する．図 5.3.2-2 に，DO ループの並列化を行う場合の指示行の指定例を示す．

```
!$OMP parallel do
    do K=1,n
     do J=1,n
        :
     enddo
    enddo
!$OMP end parallel do
```

図 5.3.2-2 OpenMP でのループの並列化の指定例

並列化対象のループ内だけで使用される配列や変数については，スレッドごとにメモリ領域を確保する必要がある．その場合，PRIVATE 宣言を行う．PRIVATE 宣言が必要な配列，変数としては，ループ内で作業配列・変数として使用されているもの，並列化の対象となるループ変数の次元をもたない配列がある．図 5.3.2-3 に PRIVATE 宣言の指定例を示す．

```
!$OMP parallel do private(a)
    do K=1,n
     do J=1,n
       do I=1,n
         a(I,J) = b(I,J,K)*c(I,K)
         d(I,J,K) = d(I,J,K)+a(I,J)*e(J,K)
       enddo
     enddo
    enddo
!$OMP end parallel do
```

図 5.3.2-3 OpenMP での PRIVATE 宣言の指定例

総和計算を行っているループを並列化する場合，スレッドごとに部分和を計算し，最後に部分和

を足し合わせて総和を求める必要がある．全スレッドが共通のメモリ領域に対して部分和の計算結果を書き込んでしまう場合，正しい結果を得ることはできない．そこで，スレッドごとに部分和を格納する領域を用意する必要がある．このような演算を「リダクション演算」と呼び，リダクション演算が必要な箇所をコンパイラに指示する必要がある．図 5.3.2-4 に，リダクション演算の指定例を示す．

```
!$OMP parallel do reduction(+:sum)
    do I=1,n
     sum=sum+I
    enddo
!$OMP end parallel do
```

図 5.3.2-4 OpenMP でのリダクション演算の指定例

リダクション演算の指定方法は以下の通り．

reduction (オペレータ:変数名リスト)

オペレータとしては，+ (和)のほかに，- （差），* （積），MAX （最大値），MIN （最小値)などが指定可能である．

5.3.3. 分散メモリ並列

分散メモリ並列とは，図 5.3.3-1 のように，ネットワークを介して接続されている複数の独立したメモリ空間を持つ複数ノードで行われる並列処理のことである．分散メモリ並列を表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 は分散メモリ環境の並列実行におけるメッセージパッシングの標準規格であり，複数のプロセス間でデータをやり取りするために用いるメッセージ通信操作の仕様標準である．FORTRAN，C 言語から呼び出すサブプログラムライブラリであり，標準化されたライブラリインタフェースにより，様々な MPI 実装環境でプログラムを修正することなくコンパイル・実行することができる．独立したメモリ空間を持つノードを複数使用してプログラムを実行するため，共有メモリ並列に比べて，大きなメモリ空間を使用した大規模なプログラムを実行することができる．一方で，プログラムを MPI 化するためには，プログラムを分析し，データや処理を分割し，通信の内容とタイミングをユーザが指定する必要があるため，ユーザの負担は大きくなる．

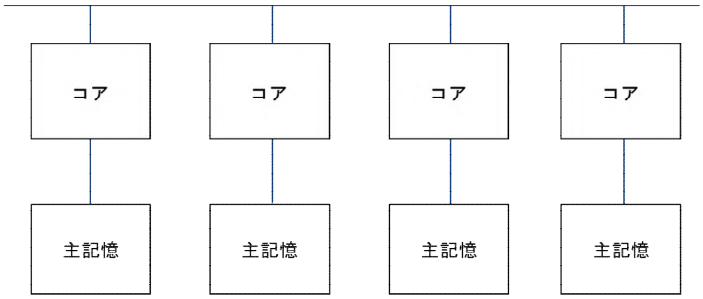


図 5.3.3-1 分散メモリ並列

(1) 処理の分割

分散並列化を行う場合，最初に並列化の対象が何であるかの検討が必要である．シミュレーションの対象となる物理現象に空間的な並列性がある場合には，領域分割法による並列化が可能である．また，複数の現象から構成される物理現象のシミュレーションにおいて，それぞれの現象を並列に処理できるような場合，現象の並列性があるといい，この場合には処理の並列化が可能になる．

図 5.3.3-2 に，空間的な並列性がある場合の，処理の分割イメージを示す．シミュレーションの対象となる領域を空間 A～空間 D に分割し，複数のプロセス(コア)で分担して空間ごとに処理 1～処理 n を実行する．1 つのプロセスで全空間の処理を行う場合に比べて，複数のプロセスで分担して処理を行うことで実行時間が短縮される．高い並列性能を実現するためには，各プロセスに割り当てられた処理をなるべく均等にする必要がある．領域分割を行った場合，分割した空間の境界面で，隣接する領域のデータを参照することが必要になる場合がある．この場合，MPI ライブラリを使用してデータの転送を行う．データの転送については「5.3.3. (2) データ転送」に記載する．

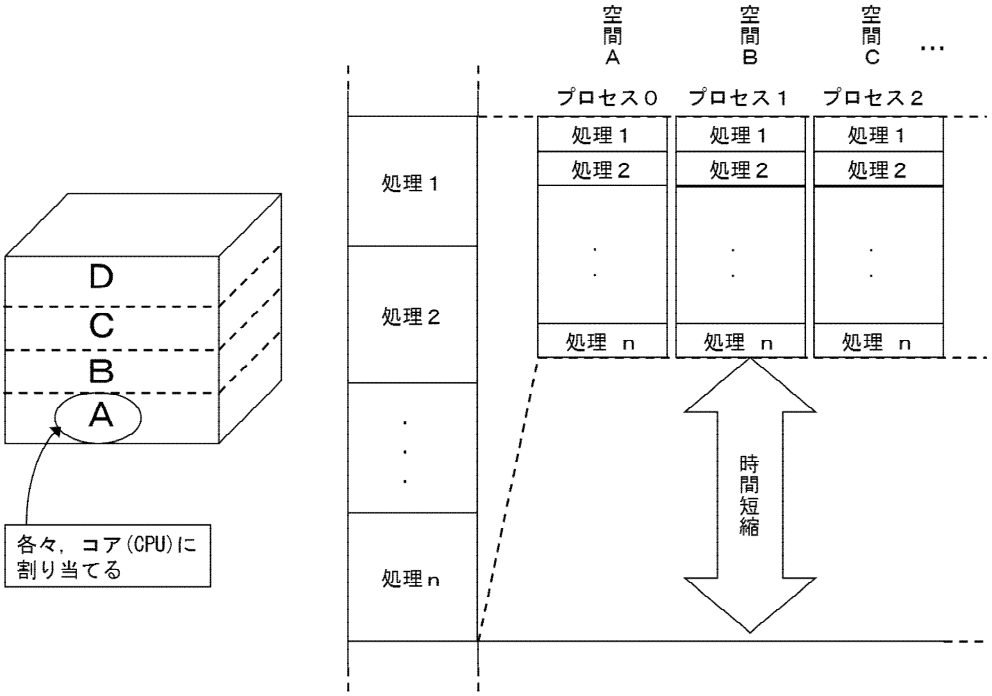


図 5.3.3-2 空間的な並列性がある場合の処理分割概念図

図 5.3.3-3 に，現象の並列性がある場合の処理の分割イメージを示す．プログラムの中で処理 2～処理 5 が，それぞれ独立して計算可能な現象の計算を行っている場合，処理 2～処理 5 を複数のプロセスで分担して処理を行うことで実行時間が短縮される．

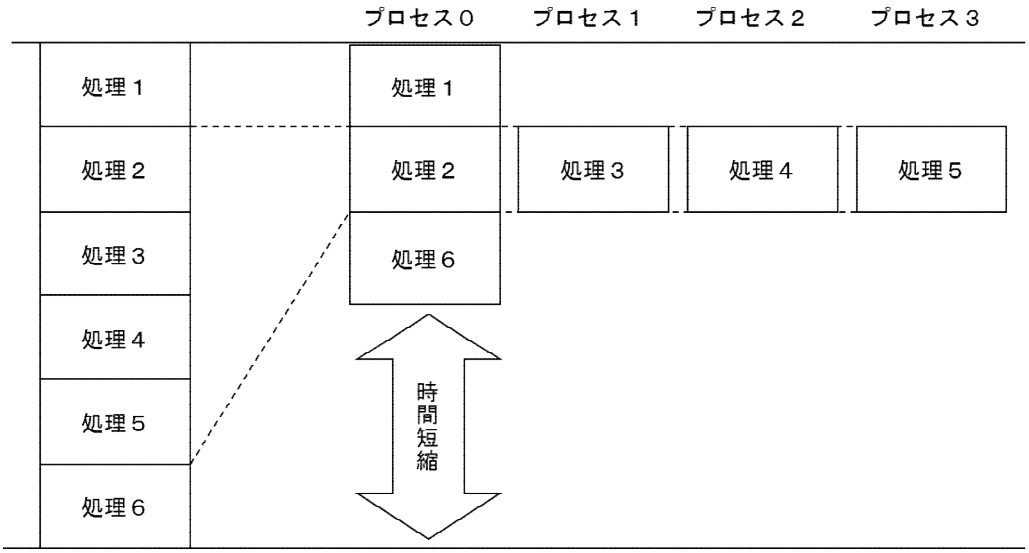


図 5.3.3-3 現象の並列性がある場合の処理分割概念図

(2) データ転送

領域分割法では，演算に必要なデータが自プロセスの担当する領域外（他プロセスの担当領域）にある場合，そのデータが含まれる領域を担当するプロセスからデータを受け取る必要がある．MPI には，様々なデータ転送方法が提供されている．

図 5.3.3-4 に，領域の境界面でデータの転送が必要な場合のイメージ図を示す．配列 a(1)～a(5) はプロセス 0 が担当する領域のデータであり，配列 a(6)～a(10) は，プロセス 1 が担当する領域のデータである．プロセス 1 の計算において隣接する領域の境界面のデータ a(5) が必要になる場合，プロセス 0 から a(5) のデータを受け取る必要があり，このような通信には 1 対 1 通信を使用する．1 対 1 通信とは，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る．MPI では 1 対 1 通信として，同期型の通信である MPI_SEND，MPI_RECV，非同期型の通信である MPI_ISEND，MPI_IRECV が利用可能である．

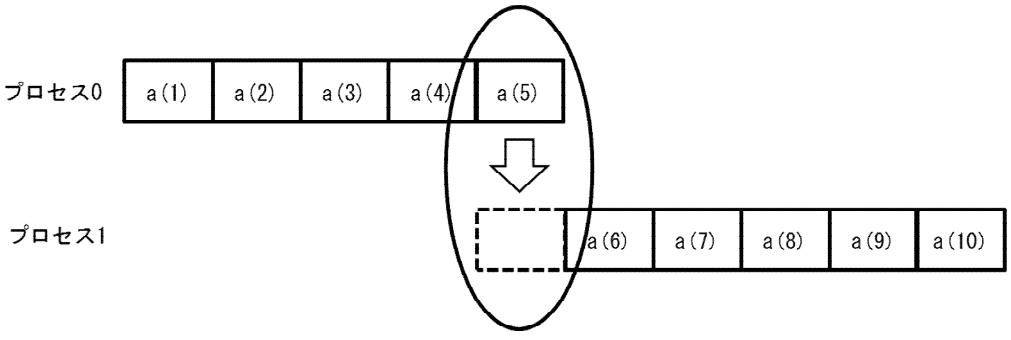


図 5.3.3-4 領域境界面でのデータ転送の概念図

1 対 1 通信が一組の送信プロセスと受信プロセス間でのデータの転送であるのに対し，全プロセスで行う同期的通信を集団通信と呼ぶ．集団通信には，データを各プロセスに配る MPI_SCATTER，デ

ータを各プロセスから集める MPI_GATHER，総和や MAX/MIN などの処理を行うリダクション処理（MPI_REDUCE）などがある．

5.4. 性能解析ツール

プログラムを高速に実行するためには，プログラムの最適化状況を把握し，ベクトル演算率，平均ベクトル長，バンクコンフリクト時間といった主要な性能指標が，十分な性能値になっているかを確認する必要がある．SX-ACE では，様々な性能分析ツールを利用し性能分析を実施することができる．図 5.4-1 に SX-ACE での最適化の流れとそれぞれのフェーズで性能分析を実施するのに有効な性能解析ツールを記載する．

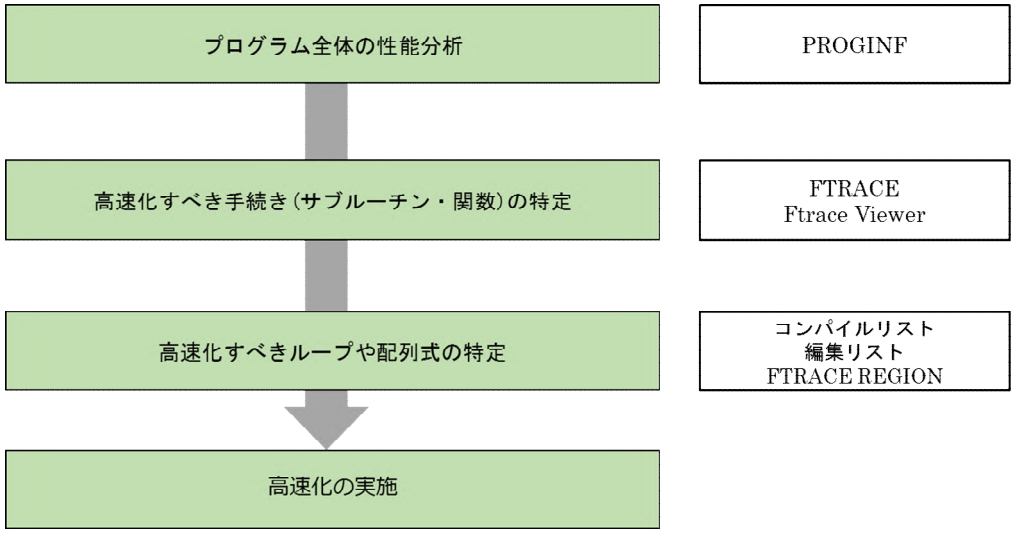


図 5.4-1 最適化の流れと各フェーズで有効な性能解析ツール

5.4.1. コンパイルリスト

コンパイラは，最適化，ベクトル化，並列化の実施状況をコンパイルリストとしてコンパイル時に出力する．図 5.4.1-1 にコンパイルリストの出力例を記載する．この例では，ソースプログラム 7 行目のループはベクトル化されているが，18 行目のループはベクトル化されていないことがわかる．

```
f90: vec(1): sample1.f, line 7: ループ全体をベクトル化する。
f90: vec(3): sample1.f, line 18: ベクトル化できないループである。
f90: sample1.f, _MAIN: There are 2 diagnoses.
```

図 5.4.1-1 コンパイルリストの出力例

コンパイルオプション「-Wf,-pvctl fullmsg」を指定することで，より詳細な情報を出力させることができる．図 5.4.1-2 に，詳細なコンパイルリストの出力例を記載する．

```
f90: opt(1592): sample1.f, line 6: 外側ループのアンローリングを行った。
f90: vec(1): sample1.f, line 7: ループ全体をベクトル化する。
f90: vec(29): sample1.f, line 7: 配列ｃに対して ADB を使用する。
f90: vec(29): sample1.f, line 7: 配列ｂに対して ADB を使用する。
f90: vec(29): sample1.f, line 7: 配列ａに対して ADB を使用する。
f90: vec(3): sample1.f, line 18: ベクトル化できないループである。
f90: opt(1017): sample1.f, line 19: サブルーチン呼出しがあるため最適化できない。
f90: vec(10): sample1.f, line 19: ループまたは配列式全体をベクトル化不可とする手続 sub が指定された。
f90: sample1.f, _MAIN: There are 8 diagnoses.
```

図 5.4.1-2 詳細なコンパイルリストの出力例

詳細なコンパイルリストを確認することで，18 行目のループがベクトル化できない原因はサブルーチン呼び出しがあるためであることが確認できる．

5.4.2. プログラム実行解析情報(PROGINF)

プログラム終了時にプログラム実行解析情報(PROGINF)が標準エラー出力ファイルに出力される．PROGINF にはプログラム全体の性能情報が出力され，ベクトル演算率，平均ベクトル長，バンクコンフリクト時間といった主要な性能指標がどのくらいの性能値になっているかを確認することができる．図 5.4.2-1 に PROGINF の出力例を記載する．

```
      ******  Program Information  ******
①  Real Time (sec)                :          48.700479
②  User Time (sec)                :          48.698599
③  Sys  Time (sec)                :           0.002122
④  Vector Time (sec)              :          48.698337
⑤  Inst. Count                    :       20615981614.
⑥  V. Inst. Count                 :       11113776147.
⑦  V. Element Count               :     2822896173772.
⑧  V. Load Element Count          :     1290401280052.
⑨  FLOP Count                     :     1290409377425.
⑩  MOPS                           :        58161.804187
⑪  MFLOPS                         :        26497.874763
⑫  A. V. Length                   :          253.999733
⑬  V. Op. Ratio (%)               :           99.664517
⑭  Memory Size (MB)               :          384.031250
⑮  MIPS                           :          423.338290
⑯  I-Cache (sec)                  :            0.000099
⑰  O-Cache (sec)                  :            0.000147
    Bank Conflict Time
⑱    CPU Port Conf. (sec)         :            3.895059
⑲    Memory Network Conf. (sec)   :           13.107002
⑳  ADB Hit Element Ratio (%)      :           46.149240
```

図 5.4.2-1 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
② User Time	ユーザ時間(ユーザルーチン実行に要した時間)
③ Sys Time	システム時間(システムルーチン実行に要した時間)
④ Vector Time	ベクトル命令実行時間
⑤ Inst. Count	全命令実行数
⑥ V. Inst. Count	ベクトル命令実行数

⑦ V. Element Count	ベクトル命令実行要素数
⑧ V. Load Element Count	ベクトル命令ロード要素数
⑨ FLOP Count	浮動小数点データ実行要素数
⑩ MOPS	1 秒間に実行された演算数を 100 万単位で示した値
⑪ MFLOPS	1 秒間に実行された浮動小数点演算数を 100 万単位で示した値
⑫ A. V. Length	平均ベクトル長
⑬ V. OP. RATIO	ベクトル演算率
⑭ Memory Size	メモリ使用量
⑮ MIPS	1 秒間に実行された命令数を 100 万単位で示した値
⑯ I-Cache	命令キャッシュミスによる発生した合計時間(秒)
⑰ O-Cache	オペランドキャッシュミスにより発生した合計時間(秒)
⑱ CPU Port Conf.	CPU ポート競合時間(秒)
⑲ Memory Network Conf.	メモリネットワーク競合時間(秒)
⑳ ADB Hit Element Ratio	ベクトル命令によりロードされた要素のうち，ADB にバッファリングされたデータがロードされた要素の割合

5.4.3. 編集リスト

編集リストは，ベクトル化や自動並列化に関する情報をソースプログラムの左側に表示したリストであり，どのループがベクトル化，自動並列化されたかを確認することができる．コンパイル時に，「-R5」オプションを指定することで，「ソースプログラム名.L」というファイル名で作成される．図 5.4.3-1 に編集リストの出力例を示す．行番号とソースプログラムの間にベクトル化や自動並列化に関数する情報が表示される．

```
FILE NAME: sample.f
PROGRAM NAME: sub
FORMAT LIST

  LINE    LOOP      FORTRAN STATEMENT

    11:
    12:                     subroutine sub(a,b,c)
    13:                     real a(100),b(100),c(100)
    14:
    15: V------>            do I=1,100
    16: |        I            call sub2(a,b,c)
    17: |        A            a(I) = b(I) + c(I)
    18: |        S            print *, a(I)
    19: V------             enddo
```

図 5.4.3-1 編集リストの出力例

以下に編集リストの主な表示例を記載する．

(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=1,100
|                   a(I)=b(I)+c(I)
V------           enddo
```

図 5.4.3-2 ループ全体がベクトル化された場合の編集リスト

(2) ベクトル化されない場合

ベクトル化されていないループには“+”が表示される.

```
+------>          do I=1,100
|                   print *, a(I)
+------           enddo
```

図 5.4.3-3 ベクトル化されない場合の編集リスト

(3) ループの一部がベクトル化された場合

ベクトル化不可の処理がある行には“S”が表示される.

```
V------>          do I=1,100
|                   a(I)=b(I)+c(I)
|       S           print *, a(I)
V------           enddo
```

図 5.4.3-4 ループの一部がベクトル化される場合の編集リスト

(4) 配列に対して ADB が使用された場合

ADB を使用したベクトルロード・ストアがある行に“A”が表示される.

```
+------>          do J=1,100
|V----->           do I=1,100
||     A             a(I,J)=a(I,J+1)+b(I,J)
|V-----            enddo
+------           enddo
```

図 5.4.3-5 配列に対して ADB が使用される場合の編集リスト

(5) 配列式をベクトル化した場合

ループの先頭と最後の行が同じである場合, ループの構造は“=”で表示される.

```
                  real a(100),b(100),c
V======           a(1:100)=b(1:100)+c
```

図 5.4.3-6 配列式をベクトル化した場合の編集リスト

ループ融合している場合は, その範囲について“V”で表示される.

```
                   real a(100),b(100),c
                   integer d(100), e(100)
 V------>          a(1:100)=b(1:100)+c
 |                 d(1:100)=int(a(1:100))
 V------           e(1:100)=d(1:100)+1
```

図 5.4.3-7 配列式がループ融合している場合の編集リスト

(6) 手続き呼び出しがインライン展開された場合

インライン展開された手続きがある行には“I”が表示される.

```
I            call sub(x,a,b,c)
```

図 5.4.3-8 手続き呼び出しがインライン展開された場合の編集リスト

(7) 多重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 J=1,100
|*----->           do I=1,100
||                    a(I,J)=b(I,J)+c(I,J)
|*-----             enddo
W------            enddo
```

図 5.4.3-9 多重ループが一重化された場合の編集リスト

(8) ループの入れ換えが行われた場合

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, ベクトル化されなくなるループには“+”が表示される.

```
X------>          do J=1,1000
|+----->           do I=1,10
||                    a(I,J)=b(I,J)+c(I,J)
|+-----             enddo
X------            enddo
```

図 5.4.3-10 ループの入れ換えが行われた場合の編集リスト

(9) ベクトル化と並列化された場合

ベクトル化と並列化が行われたループに“Y”が表示される.

```
Y------>          do I=1,10000
|                   a(I)=a(I)+b(I)*c(I)
Y------           enddo
```

図 5.4.3-11 ベクトル化と並列化された場合の編集リスト

(10) 並列化された場合

並列化が行われたループに“P”が表示される.

```
P------>          do J=1,100
|V----->           do I=1,100
||                    a(I,J)=b(I,J)+c(I,J)
|V-----             enddo
P------            enddo
```

図 5.4.3-12 並列化された場合の編集リスト

(11) 複数の情報がある場合

一行に対して複数の情報がある場合, “M”が表示される.

※配列 a はベクトル化され, 配列 b はループ長が短いためループが展開されるので, この一行には複数の情報がある.

```
M======           a(1:10000)=0.0d0 ; b(1:3)=0.0d0
```

図 5.4.3-13 複数の情報がある場合の編集リスト

5.4.4.　簡易性能解析機能（FTRACE）

コンパイル時にオプション「-ftrace」を指定することで，簡易性能解析情報を採取することができる．簡易性能解析情報には，手続き（サブルーチン）ごとに実行時間，ベクトル化率，平均ベクトル長，バンクコンフリクト時間などの情報が含まれており，十分に性能値が得られていない手続きを特定することができる．実行時間が長く，性能値が低い手続きをターゲットにしてチューニングを実施することで，プログラムの高速化を効率よく実施することができる．図 5.4.4-1 に簡易性能解析機能による解析リストの例を示す．

```
   ①         ②          ③                ④           ⑤        ⑥      ⑦     ⑧         ⑨       ⑩       ⑪           ⑫              ⑬
PROC.NAME  FREQUENCY  EXCLUSIVE         AVER.TIME     MOPS   MFLOPS V.OP  AVER.     VECTOR I-CACHE O-CACHE    BANK CONFLICT      ADB HIT
                      TIME[sec]( % )     [msec]                     RATIO V.LEN       TIME    MISS    MISS CPU PORT  NETWORK    ELEM.%

sub1              10   285.153( 95.9) 28515.315   3895.0   1807.4 96.68  64.0    285.149   0.000   0.003    0.000  107.370     11.77
sub2               3    12.161(  4.1)  4053.827  14369.4   4237.9 95.85  64.0     12.161   0.000   0.000    0.000    7.484     20.06
main_              1     0.045(  0.0)    44.809  45681.9      0.0 99.62 250.7      0.045   0.000   0.000    0.000    0.000     89.98
------------------------------------------------------------------------------------------------------------------------------------
total             14   297.359(100.0) 21239.960   4329.6   1906.6 96.57  64.1    297.355   0.000   0.003    0.000  114.854     12.95
```

図 5.4.4-1　解析リストの出力例

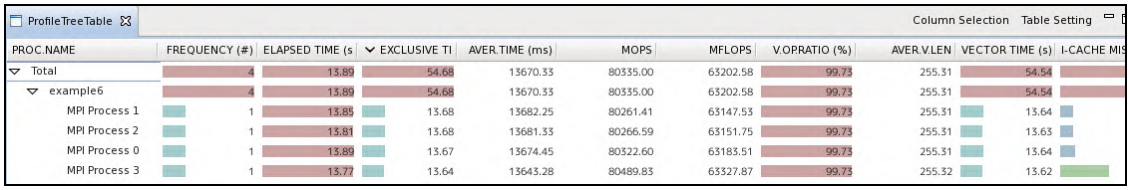
表示される各項目の意味は以下の通り．

① PROC.NAME	手続き名
② FREQUENCY	手続きの呼び出し回数
③ EXCLUSIVE TIME	手続きの実行に要した占有の CPU 時間（秒）と，手続き全体の実行に要した CPU 時間に対する比率
④ AVER.TIME	手続きの 1 回の実行に要した平均 CPU 時間（ミリ秒）
⑤ MOPS	1 秒間に実行された演算数を 100 万単位で示した値
⑥ MFLOPS	1 秒間に実行された浮動小数点演算数を 100 万単位で示した値
⑦ V.OP RATIO	ベクトル演算率
⑧ AVER V.LEN	平均ベクトル長
⑨ VECTOR TIME	ベクトル命令実行時間（秒）
⑩ I-CACHE MISS	命令キャッシュミスによる発生した合計時間（秒）
⑪ O-CACHE MISS	オペランドキャッシュミスにより発生した合計時間（秒）
⑫ BANK CONFLICT	バンクコンフリクト時間（秒）
CPU PORT	CPU ポート競合時間（秒）
NETWORK	メモリネットワーク競合時間（秒）
⑬ ADB.HIT ELEM	ベクトル命令によりロードされた要素のうち，ADB にバッファリングされたデータがロードされた要素の割合

5.4.5.　Ftrace Viewer

SX-ACE では，視覚的にプログラムの性能を分析するための性能解析ツールである Ftrace Viewer を使用することができる．Ftrace Viewer では，手続きごとの実行回数，実行時間，ベクトル演算率，平均ベクトル長，バンクコンフリクト時間などの様々な性能値に加えて，それをグラフ表示することで視覚

的に性能分析を行うことができる. 図 5.4.5-1 に MPI プログラムを 4MPI プロセスで実行した場合の性能値の表示例(Profile Tree Table)を示す. MPI プロセスごとに性能値が表示されている.



PROC.NAME	FREQUENCY (#)	ELAPSED TIME (s	EXCLUSIVE TI	AVER.TIME (ms)	MOPS	MFLOPS	V.OPRATIO (%)	AVER.V.LEN	VECTOR TIME (s)	I-CACHE MIS
Total	4	13.89	54.68	13670.33	80335.00	63202.58	99.73	255.31	54.54	
example6	4	13.89	54.68	13670.33	80335.00	63202.58	99.73	255.31	54.54	
MPI Process 1	1	13.85	13.68	13682.25	80261.41	63147.53	99.73	255.31	13.64	
MPI Process 2	1	13.81	13.68	13681.33	80266.59	63151.75	99.73	255.31	13.63	
MPI Process 0	1	13.89	13.67	13674.45	80322.60	63183.51	99.73	255.31	13.64	
MPI Process 3	1	13.77	13.64	13643.28	80489.83	63327.87	99.73	255.32	13.62	

図 5.4.5-1 Profile Tree Table の表示例

“MPI Communication Chart”モードを使用することで, MPI プログラムの MPI 通信時間に関する性能情報を視覚的に確認することができる. 図 5.4.5-2 に, ”MPI Communication Chart”モードの表示例を示す. 縦軸に実行時間, 横軸に MPI プロセス番号が表示されており, プロセスごとに実行時間, MPI 通信時間, MPI のアイドル時間が積み上げグラフとして表示される. 図 5.4.5-2 の例では, MPI 通信もバランスよく実行されており, MPI のアイドル時間もほとんど発生してないことがわかる. 大規模なプログラムを実行する場合, MPI プロセス数も膨大になる. Ftrace Viewer を使用することで, 演算や通信のインバランスがどの MPI プロセスで発生しているのかを容易に把握することができる.

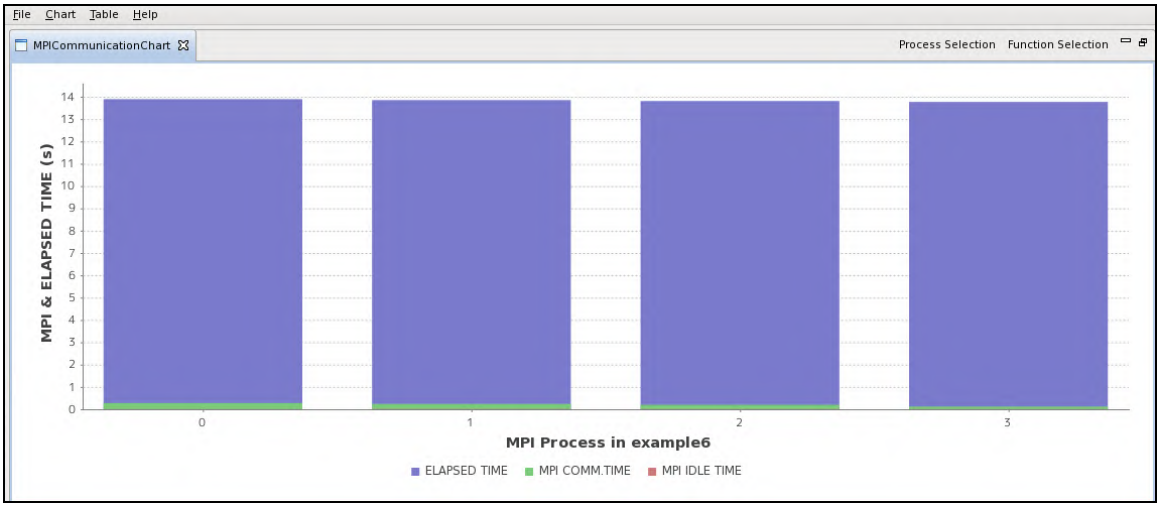


図 5.4.5-2 “MPI Communication Chart”モードの表示例

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　江川隆輔
情報部情報基盤課　大泉健治　小野敏　山下毅　齋藤敦子　佐々木大輔　森谷友映
高性能計算技術開発(NEC)共同研究部門　撫佐昭裕　渡部修　曽我隆　山口健太　下村陽一
NEC ソリューションイノベータ株式会社　坂口祐太

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)	ベクトル長が短い場合
6.1.2	ループ展開による高速化(2)	ベクトル長が短い場合
6.1.3	ループの一重化による高速化	ベクトル長が短い場合
6.1.4	ループ融合による高速化(1)	構造が同じ多重ループが連続している場合
6.1.5	ループ融合による高速化(2)	構造が同じ多重ループが連続している場合
6.1.6	ループ分割による高速化 (巨大ループの分割)	ベクトル演算率が低い場合
6.1.7	IF 文のループ外への移動による 高速化	ベクトルループ内に IF 文が含まれる場合
6.1.8	RedBlack 法のメモリアクセス方法変更 による高速化	間接参照がある場合
6.1.9	ハイパープレーン法のアクセス パターン変更による高速化	ADB ヒット率が低い場合
6.1.10	作業配列の変数化によるメモリアクセス 負荷低減による高速化	作業配列の容量が大きい場合
6.1.11	命令の追い越しによる高速化	間接参照がある場合
6.1.12	ADB によるメモリアクセス低減による 高速化	ADB ヒット率が低い場合
6.1.13	リストアクセスの削減による高速化	間接参照がある場合
6.1.14	外側ループのアンロールによる メモリアクセス効率化	メモリネットワーク競合時間が大きい場合
6.1.15	配列のサイズ変更による バンクコンフリクト時間の削減	メモリネットワーク競合時間が大きい場合
6.1.16	作業配列の導入による I/O 時間の短縮	ファイル I/O を高速化する場合
6.1.17	ファイル形式の変更による I/O の高速化	ファイル I/O を高速化する場合
6.1.18	複合問題の事例	ベクトル長が短い場合 ベクトル演算率が低い場合
6.2.1	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
6.3.1	メモリ使用量の削減	MPI プログラムでメモリ使用量を削減する場合
6.3.2	リダクション処理の変更による高速化	MPI プログラムで総和や最大値を求める場合
6.3.3	分散 I/O による I/O の効率化	MPI プログラムでファイル I/O を高速化する場合
6.3.4	転置通信処理のシフト通信処理への 変更による高速化	MPI プログラムで転置通信処理に時間がかかる 場合

6.1. SX-ACE のノード内最適化事例

6.1.1. ループ展開による高速化(1)

(1) チューニング方針

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 にチューニング前のコードを示す．三重 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 に示したチューニング前の FTRACE 情報では，平均ベクトル長が 27.9 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 i を持つ最内ループの繰り返し数（nx-lnm～nx1）が 13 と短いためである．最内ループを展開する EXPAND 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．

```
!チューニング前
 +------>    do k = 1,nz1
 |+----->      do j = 1,ny1
 ||V---->       do i= nx-lnm,nx1                                  ループ長：13
 |||     A        a(i-(nx1-lnm),j,k,2) = b(j)*a(i-(nx1-lnm),j,k,2) &
 |||                                          -c(j)*(d(i,j+1,k)-d(i,j,k))
 |||     A        e(i-(nx1-lnm),j,k,2) = f(k)*e(i-(nx1-lnm),j,k,2) &
 |||                                          +g(k)*(h(i,j,k+1)-h(i,j,k))
 |||     A        i(i,j,k) = a(i-(nx1-lnm),j,k,2)+e(i-(nx1-lnm),j,k,2)
 |||
 |||     A        j(i-(nx1-lnm),j,k,2) = f(k)*j(i-(nx1-lnm),j,k,2) &
 |||     A                                    -g(k)*(k(i,j,k+1)-k(i,j,k))
 |||     A        l(i-(nx1-lnm),j,k,2) = m(i)*l(i-(nx1-lnm),j,k,2) &
 |||                                          +n(i)*(d(i+1,j,k)-d(i,j,k))
 |||     A        o(i,j,k) = j(i-(nx1-lnm),j,k,2)+l(i-(nx1-lnm),j,k,2)
 |||     A
 |||     A        p(i-(nx1-lnm),j,k,2) = m(i)*p(i-(nx1-lnm),j,k,2) &
 |||                                          -n(i)*(h(i+1,j,k)-h(i,j,k))
 |||     A        q(i-(nx1-lnm),j,k,2) = b(j)*q(i-(nx1-lnm),j,k,2) &
 |||                                          +c(j)*(k(i,j+1,k)-k(i,j,k))
 |||     A        r(i,j,k) = p(i-(nx1-lnm),j,k,2)+q(i-(nx1-lnm),j,k,2)
 ||V----         end do
 |+-----       end do
 +------     end do
```

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	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
1000	19.689(6.2)	19.689	15095.9	2838.1	97.26	**27.9**	19.689	0.000	0.000	0.000	9.285	33.14	【チューニング前】

図 6.1.1-2 ループ展開前の FTRACE 情報

(2) チューニング内容

図 6.1.1-3 にチューニング後のコードを示す．最内ループのループ長については，本ループよりも前で，nx1=nx+1，lnm=13 と定義されコンパイル時に判明しているので，EXPAND 指示行にループ長 lnm 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．

```
!チューニング後
 +------>    do k = 1,nz1
 |V----->      do j = 1,ny1
 ||          !cdir expand=lnm          <-- ループ展開
 ||*---->        do i= nx-lnm,nx1
 |||     A         a(i-(nx1-lnm),j,k,2) = b(j)*a(i-(nx1-lnm),j,k,2) &
 |||                                        -c(j)*(d(i,j+1,k)-d(i,j,k))
 |||     A         e(i-(nx1-lnm),j,k,2) = f(k)*e(i-(nx1-lnm),j,k,2) &
 |||                                        +g(k)*(h(i,j,k+1)-h(i,j,k))
 |||     A         i(i,j,k) = a(i-(nx1-lnm),j,k,2)+e(i-(nx1-lnm),j,k,2)
 |||
 |||     A         j(i-(nx1-lnm),j,k,2) = f(k)*j(i-(nx1-lnm),j,k,2) &
 |||     A                                  -g(k)*(k(i,j,k+1)-k(i,j,k))
 |||     A         l(i-(nx1-lnm),j,k,2) = m(i)*l(i-(nx1-lnm),j,k,2) &
 |||                                        +n(i)*(d(i+1,j,k)-d(i,j,k))
 |||     A         o(i,j,k) = j(i-(nx1-lnm),j,k,2)+l(i-(nx1-lnm),j,k,2)
 |||     A
 |||     A         p(i-(nx1-lnm),j,k,2) = m(i)*p(i-(nx1-lnm),j,k,2) &
 |||                                        -n(i)*(h(i+1,j,k)-h(i,j,k))
 |||     A         q(i-(nx1-lnm),j,k,2) = b(j)*q(i-(nx1-lnm),j,k,2) &
 |||                                        +c(j)*(k(i,j+1,k)-k(i,j,k))
 |||     A         r(i,j,k) = p(i-(nx1-lnm),j,k,2)+q(i-(nx1-lnm),j,k,2)
 ||*----         end do
 |V-----       end do
 +------     end do
```

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 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
1000	19.689(6.2)	19.689	15095.9	2838.1	97.26	**27.9**	19.689	0.000	0.000	0.000	9.285	33.14	【チューニング前】
1000	3.458(2.0)	3.458	35005.1	16159.3	99.80	**199.7**	3.458	0.000	0.000	0.098	0.620	87.06	【チューニング後】

図 6.1.1-4 ループ展開前後の FTRACE 情報

EXPAND 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 27.9 から 199.7 に増加し, 実行時間が 19.7 秒から 3.5 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) チューニング方針

図 6.1.2-1 にチューニング前のコードを示す. 本事例もベクトル化の対象となる DO ループのベクトル長が短く, その外側の DO ループでベクトル化を行うものである. ただし, 本事例は 6.1.1 のように EXPAND 指示行が利用できない場合の例である.

本コードは三重 DO ループ構造と配列式からなっており, インデックス i1 のループがベクトル化されている. しかし, このループのループ長は 10 と非常に短い. 本事例では UNROLL 指示行を挿入してこの DO ループを展開し, その外側の DO ループでベクトル化して実行時間の短縮を図る.

図 6.1.2-2 の FTRACE 情報から, チューニング前には平均ベクトル長が 21.9 と短く, 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.

```
!チューニング前
 +------>        do k = lkbgn, lkend
 |+----->        do j = ljbgn, ljend
 ||                 a(ls1,j,k) = b(l1,j,k)
 ||                 a(ls2,j,k) = b(l2,j,k)
 ||                 a(ls3,j,k) = b(l3,j,k)
 ||                 a(ls4,j,k) = b(l4,j,k)
 ||                 a(ls5,j,k) = b(l5,j,k)
 ||                 a(ls6,j,k) = b(l6,j,k)
 ||                 a(ls7,j,k) = b(l7,j,k)
 ||
 ||                 c(ls1,j,k) = b(l1,j,k)*d(ls1)
 ||                 c(ls2,j,k) = b(l2,j,k)*d(ls2)
 ||                 c(ls3,j,k) = b(l3,j,k)*d(ls3)
 ||                 c(ls4,j,k) = b(l4,j,k)*d(ls4)
 ||                 c(ls5,j,k) = b(l5,j,k)*d(ls5)
 ||                 c(ls6,j,k) = b(l6,j,k)*d(ls6)
 ||                 c(ls7,j,k) = b(l7,j,k)*d(ls7)
  :
(省略)
  :
 ||V---->          do i1 = 1, lsp        <-- ループ長：10
 |||     A           e(ls1) = f(ls1,i1)*(1.d0+sqrt(g(ls1,i1)*h(i1)*i(ls1)))**2
 |||     A           e(ls2) = f(ls2,i1)*(1.d0+sqrt(g(ls2,i1)*h(i1)*i(ls2)))**2
 |||     A           e(ls3) = f(ls3,i1)*(1.d0+sqrt(g(ls3,i1)*h(i1)*i(ls3)))**2
 |||     A           e(ls4) = f(ls4,i1)*(1.d0+sqrt(g(ls4,i1)*h(i1)*i(ls4)))**2
 |||     A           e(ls5) = f(ls5,i1)*(1.d0+sqrt(g(ls5,i1)*h(i1)*i(ls5)))**2
 |||     A           e(ls6) = f(ls6,i1)*(1.d0+sqrt(g(ls6,i1)*h(i1)*i(ls6)))**2
 |||     A           e(ls7) = f(ls7,i1)*(1.d0+sqrt(g(ls7,i1)*h(i1)*i(ls7)))**2
 |||
 |||*=== A           j(i1) = sum(c(1:lsp,j,k)*e(1:lsp))
 |||     A           k(i1) = var1*c(i1,j,k) + var2*j(i1)
 ||V----           end do
 ||*====           l(j,k) = sum(h(1:lsp)*c(1:lsp,j,k)/j(1:lsp))
 ||*====           m(j,k) = sum(n(1:lsp)*c(1:lsp,j,k)/k(1:lsp))
 |+-----          end do
 +------          end do
```

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	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
8001	8039.993(60.0)	1004.874	2001.1	246.5	91.75	**21.9**	7239.296	0.032	234.179	0.170	0.690	97.86	【チューニング前】

図 6.1.2-2 ループ展開前の FTRACE 情報

(2) チューニング内容

図 6.1.2-3 にチューニング後のコードを示す. 本コードは UNROLL 指示行を挿入しループを展開している. 6.1.1 の事例では EXPAND 指示行を用いてループを展開したが, EXPAND 指示行は最内ループのみに有効な指示行である. 本事例ではインデックス i1 のループの内側に配列式が存在しており, 配列式の DO ループが実際の最内ループとなっている. そのため, インデックス i1 のループに EXPAND 指示行を指定しても無効となるため, UNROLL 指示行によりループを展開している.

```
!チューニング後
 +------>        do k = lkbgn, lkend
 |V----->        do j = ljbgn, ljend
 ||                 a(ls1,j,k) = b(l1,j,k)
 ||                 a(ls2,j,k) = b(l2,j,k)
```

```
 ||                a(ls3,j,k) = b(l3,j,k)
 ||                a(ls4,j,k) = b(l4,j,k)
 ||                a(ls5,j,k) = b(l5,j,k)
 ||                a(ls6,j,k) = b(l6,j,k)
 ||                a(ls7,j,k) = b(l7,j,k)
 ||
 ||                c(ls1,j,k) = b(l1,j,k)*d(ls1)
 ||                c(ls2,j,k) = b(l2,j,k)*d(ls2)
 ||                c(ls3,j,k) = b(l3,j,k)*d(ls3)
 ||                c(ls4,j,k) = b(l4,j,k)*d(ls4)
 ||                c(ls5,j,k) = b(l5,j,k)*d(ls5)
 ||                c(ls6,j,k) = b(l6,j,k)*d(ls6)
 ||                c(ls7,j,k) = b(l7,j,k)*d(ls7)
  :
(省略)
  :
 ||         !cdir unroll=10                                  <-- ループ展開
 ||*---->          do i1 = 1, lsp
 |||                 e(ls1) = f(ls1,i1)*(1.d0+sqrt(g(ls1,i1)*h(i1)*i(ls1)))**2
 |||                 e(ls2) = f(ls2,i1)*(1.d0+sqrt(g(ls2,i1)*h(i1)*i(ls2)))**2
 |||                 e(ls3) = f(ls3,i1)*(1.d0+sqrt(g(ls3,i1)*h(i1)*i(ls3)))**2
 |||                 e(ls4) = f(ls4,i1)*(1.d0+sqrt(g(ls4,i1)*h(i1)*i(ls4)))**2
 |||                 e(ls5) = f(ls5,i1)*(1.d0+sqrt(g(ls5,i1)*h(i1)*i(ls5)))**2
 |||                 e(ls6) = f(ls6,i1)*(1.d0+sqrt(g(ls6,i1)*h(i1)*i(ls6)))**2
 |||                 e(ls7) = f(ls7,i1)*(1.d0+sqrt(g(ls7,i1)*h(i1)*i(ls7)))**2
 |||
 |||*===              j(i1) = sum(c(1:lsp,j,k)*e(1:lsp))      <-- 配列式
 |||                 k(i1) = var1*c(i1,j,k) + var2*j(i1)
 ||*----           end do
 ||*==== A         l(j,k) = sum(h(1:lsp)*c(1:lsp,j,k)/j(1:lsp))
 ||*==== A         m(j,k) = sum(n(1:lsp)*c(1:lsp,j,k)/k(1:lsp))
 |V-----         end do
 +------         end do
```

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 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
8001	8039.993(60.0)	1004.874	2001.1	246.5	91.75	**21.9**	7239.296	0.032	234.179	0.170	0.690	97.86	【チューニング前】
8001	181.999(5.9)	22.747	27860.6	10920.7	99.18	**225.9**	166.131	0.023	0.538	0.061	0.483	89.06	【チューニング後】

図 6.1.2-4 ループ展開前後の FTRACE 情報

ループが展開されたことにより, 平均ベクトル長が 21.9 から 225.9 まで増加し, 実行時間が 8040.0 秒から 182.0 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) チューニング方針

コンパイラは連続にアクセスする多重 DO ループを自動で一重化し, ループ長を長くするように試みる. 本事例は, 配列の定義に pointer 文があり, コンパイラによるループの一重化が行えない事例である.

図 6.1.3-1 にチューニング前のコードを示す. complex, pointer 文と allocate 文により定義されている多次元配列は, 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. そこで, 配列を complex 文のみで定義することにより, コンパイラは配列

の 1, 2 次元目について演算範囲が連続であることが判断でき，二重ループを一重化し，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 に示すチューニング前の FTRACE 情報では，ベクトル演算率は 99% 以上と高い値となっているが，平均ベクトル長は最内ループのループ長である 64 となっており，十分な性能を出すことができていない.

```
!チューニング前
! 配列宣言部分
          integer,parameter ::n=64,wm=2048
          complex(8),pointer::a(:,:,:),b(:,:,:)
          complex(8),pointer::c(:,:,:)
 (省略)
          allocate(a(0:n-1,0:n-1,0:wm-1))
          allocate(b(0:n-1,0:n-1,0:wm-1))
          allocate(c(0:n-1,0:n-1,0:wm-1))
! 演算部分
 |||+---> do iw=0,wm-1
 ||||        var1=dble(2*iw+1)*var2*var3
 ||||+-->    do iky=0,n-1
 |||||V->      do ikx=0,n-1
 ||||||  A       a(ikx,iky,iw)=(var4*var1-d(ikx,iky))&
 ||||||           /((var4*var1-var5-c(ikx,iky,iw)) &
 ||||||           *(var4*var1-d(ikx,iky))-e(ikx,iky))&
 ||||||           -1.0d0/(var4*var1-gza)
 |||||V-       end do
 ||||+--     end do
 |||+---  end do
 |||
 |||+---> do iw=0,wm-1
 ||||        var1=dble(2*iw+1)*var2*var3
 ||||+-->    do iky=0,n-1
 |||||V->      do ikx=0,n-1
 ||||||  A       b(ikx,iky,iw)=(var4*var1-var5 &
 ||||||           -c(ikx,iky,iw)) &
 ||||||           /((var4*var1-var5-c(ikx,iky,iw)) &
 ||||||           *(var4*var1-d(ikx,iky))-e(ikx,iky))&
 ||||||           -1.0d0/(var4*var1-var6)
 |||||V-       end do
 ||||+--     end do
 |||+---  end do
```

図 6.1.3-1 ループの一重化前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
2893	81.907(11.6)	28.312	22048.5	13630.6	99.44	**64.0**	81.907	0.001	0.001	0.000	31.853	54.92	**【チューニング前】**

図 6.1.3-2 ループの一重化前の FTRACE 情報

(2) チューニング内容

図 6.1.3-3 にチューニング後のコードを示す. 配列の宣言方法を変更することでコンパイラは 1, 2 次元目の演算範囲が連続であることを判断し，ループの一重化を行う. 最内ループに＊マーク，その外側ループには W 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.

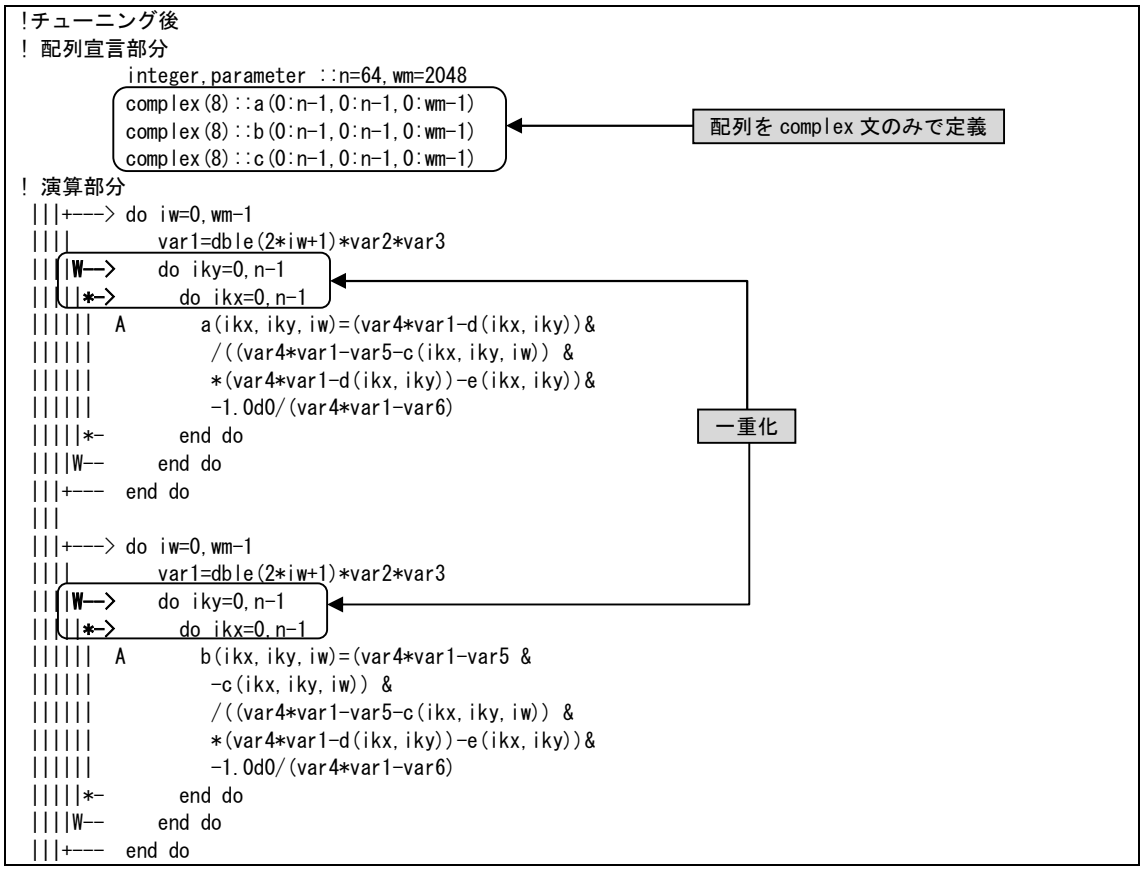


図 6.1.3-3 ループの一重化後のコード

(3) 性能分析

図 6.1.3-4 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
2893	81.907(11.6)	28.312	22048.5	13630.6	99.44	**64.0**	81.907	0.001	0.001	0.000	31.853	54.92	【チューニング前】
2893	29.389(4.7)	10.159	61210.9	37988.7	99.83	**256.0**	29.388	0.000	0.000	0.000	1.707	49.99	【チューニング後】

図 6.1.3-4 ループの一重化前後の FTRACE 情報

二重ループの一重化により, 平均ベクトル長が 64 から最大値の 256 に増加し, 実行時間が 81.9 秒から 29.4 秒に短縮されている.

6.1.4. ループ融合による高速化(1)

(1) チューニング方針

6.1.3 節のチューニング後のコードについてさらに高速化を行う. 図 6.1.4-1 が 6.1.3 節のチューニング後のコードである. 四角で囲った 2 カ所の三重 DO ループの演算範囲が同じである. 本事例では, この 2 カ所の三重 DO ループを融合し, DO ループのオーバーヘッドを削減して高速化を図る.

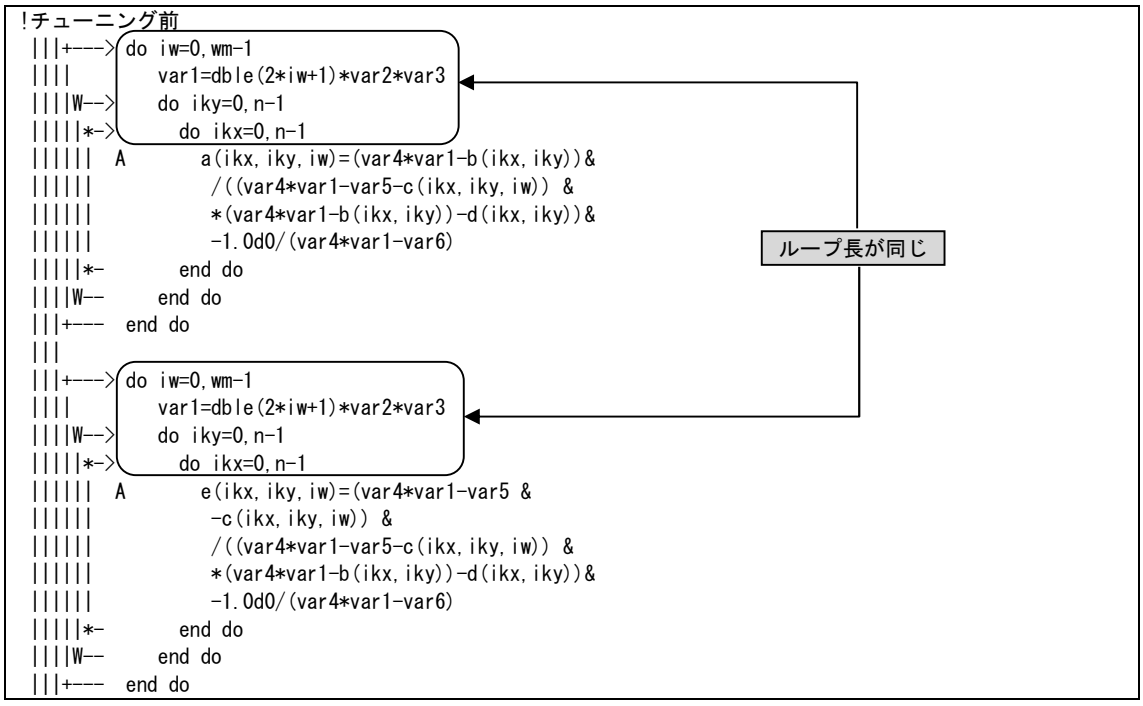


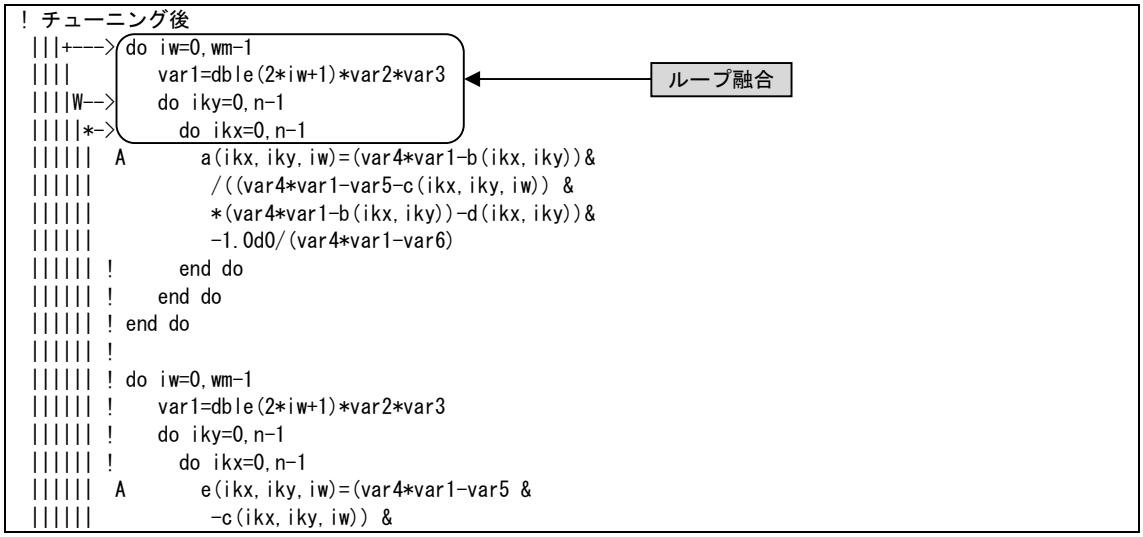
図 6.1.4-1 ループ融合前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
2893	29.389(4.7)	10.159	61210.9	37988.7	99.83	256.0	29.388	0.000	0.000	0.000	1.707	49.99	【チューニング前】

図 6.1.4-2 ループ融合前の FTRACE 情報

(2) チューニング内容

図 6.1.4-3 にチューニング後のコードを示す. 同じ演算範囲を持つ三重 DO ループの共通の演算式部分と DO 文および ENDDO 文をコメントアウトすることで, 2 カ所の三重 DO ループを一つの三重DOループに融合を行った. インデックス iw を持つ三重目のDOループの範囲にループを示す＋記号が付与され, 2 カ所の三重 DO ループが 1 つの三重 DO ループに融合されたことが確認できる.



```
 ||||||            /((var4*var1-var5-c(ikx,iky,iw)) &
 ||||||            *(var4*var1-b(ikx,iky))-d(ikx,iky))&
 ||||||            -1.0d0/(var4*var1-var6)
 |||||*-        end do
 ||||W--      end do
 |||+---   end do
```

図 6.1.4-3 ループ融合後のコード

(3) 性能分析

図 6.1.4-4 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
2893	29.389(4.7)	10.159	61210.9	37988.7	99.83	256.0	29.388	0.000	0.000	0.000	1.707	49.99	【チューニング前】
2893	20.193(3.3)	6.980	67385.5	39661.6	99.87	256.0	20.193	0.000	0.000	0.000	0.854	49.95	【チューニング後】

図 6.1.4-4 ループ融合前後の FTRACE 情報

2 カ所の三重 DO ループを 1 つの三重 DO ループにするループ融合により, ベクトル演算率, 平均ベクトル長ともに変化はないが, ベクトル演算のオーバーヘッド削減により実行時間が 29.4 秒から 20.2 秒に短縮されている.

6.1.5. ループ融合による高速化(2)

(1) チューニング方針

本事例はループ融合を行い, DO ループのオーバーヘッドを削減するとともに, 配列データのロード数を削減し, 高速化を行う例である.

図 6.1.5-1 にチューニング前のコードを示す. ループの始点と終点が同じ三重 DO ループが連続している. これらのループ間にデータの依存関係がないので, 三重 DO ループを融合することができる. また, 各多重ループ内では同じ配列 a のデータが参照されている. ループ融合を行うことにより同じ配列 a のデータをロードする必要がなくなり, メモリアクセス回数を削減することができる. 図 6.1.5-2 にチューニング前の FTRACE 情報を示す.

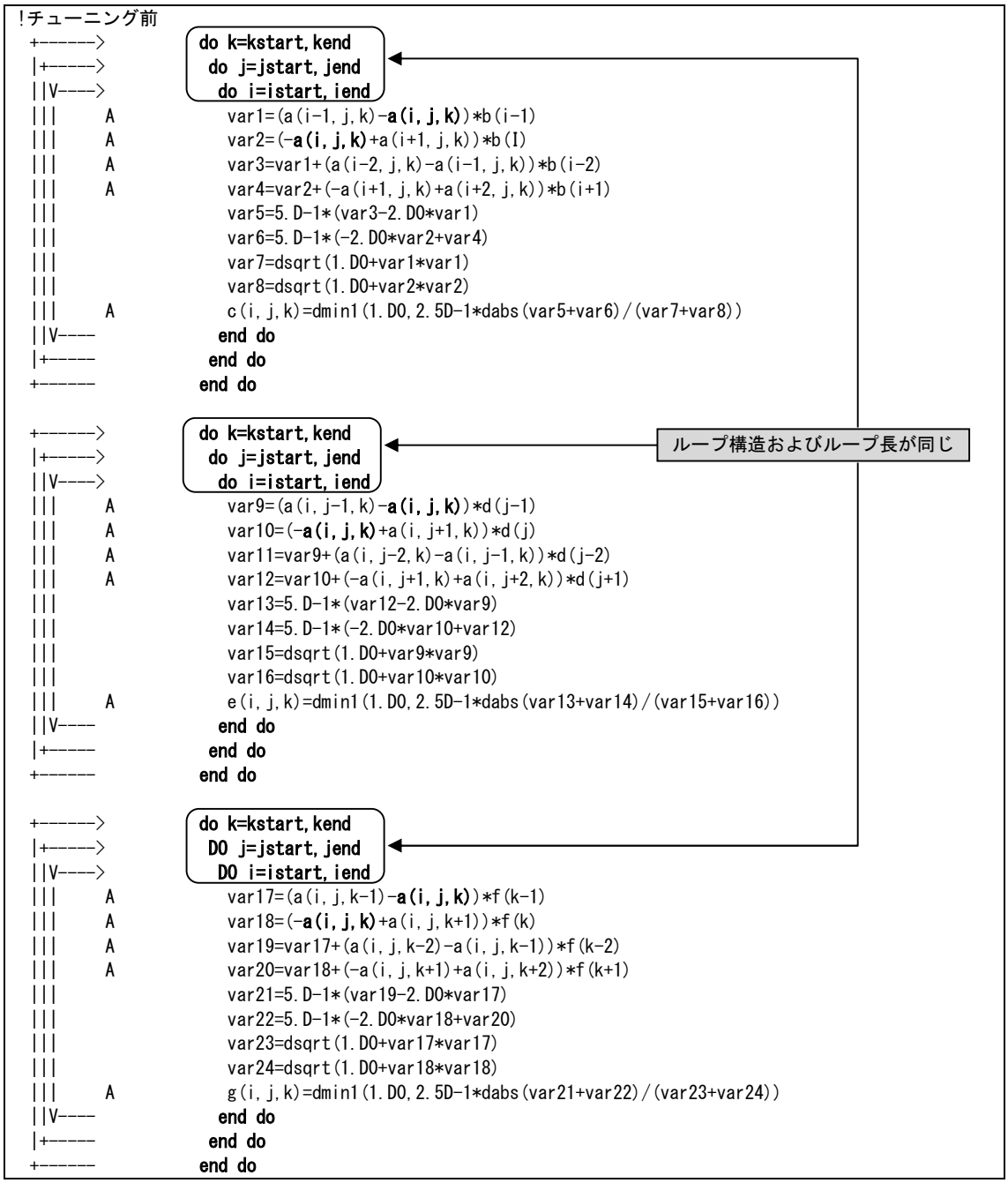


図 6.1.5-1 ループ融合前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
3546	636.604(4.6)	179.527	22862.9	9240.3	99.29	256.0	582.132	0.003	0.002	0.000	26.240	43.39	【チューニング前】

図 6.1.5-2 ループ融合前の FTRACE 情報

(2) チューニング内容

図 6.1.5-3 にチューニング後のコードを記載する. DO・ENDDO 文を削除し, 連続する DO ループを融合する.

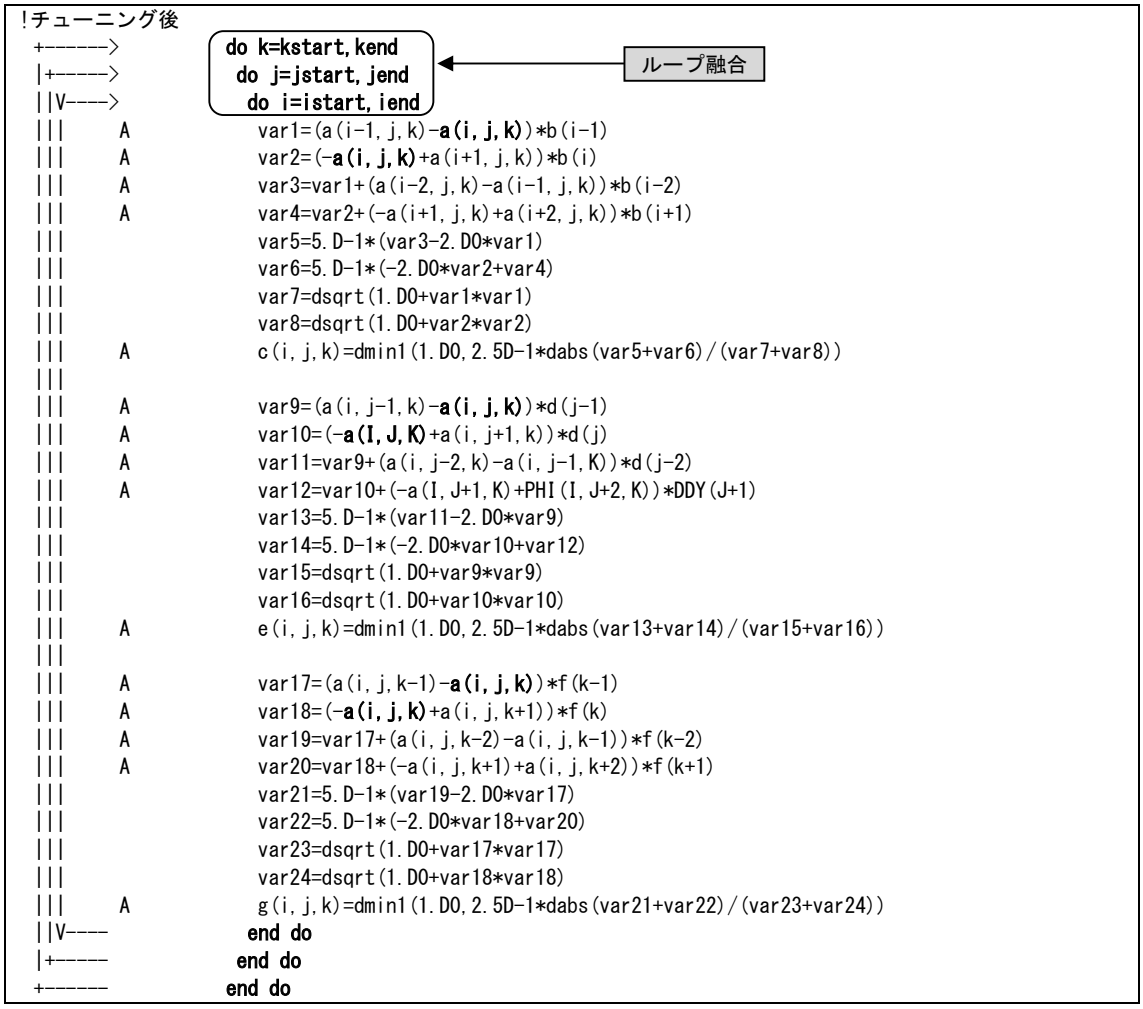


図 6.1.5-3 ループ融合後のコード

(3) 性能分析

図 6.1.5-4 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
3546	636.604(4.6)	179.527	22862.9	9240.3	99.29	256.0	582.132	0.003	0.002	0.000	26.240	43.39	【チューニング前】
3546	573.425(5.2)	161.711	25875.0	10258.3	99.36	256.0	519.373	0.001	0.002	0.000	4.542	70.87	【チューニング後】

図 6.1.5-4 ループ融合前後の FTRACE 情報

DO ループを融合することによるベクトル化のオーバーヘッド削減と配列 a のロード数の削減により, 実行時間が 636.6 秒から 573.4 秒に短縮されている.

6.1.6. ループ分割による高速化(大規模ループの分割)

(1) チューニング方針

本事例は, ベクトル化の対象となる DO ループ内において処理量が大きく, コンパイラがベクトル化と最適化の判断ができない例である.

図 6.1.6-1 にチューニング前のコードを記載する. 最内ループにはベクトル化を阻害する要因

はないが，コンパイラによるベクトル化と最適化が行われていない部分がある．この DO ループには複雑な分岐処理が多数あり，また，ループ内での処理量が多いため，コンパイラが依存関係を解析できず部分ベクトル化となっている．コンパイラが依存関係を解析できない場合には，コンパイルメッセージに「”<変数名>”は依存関係の解析ができないためベクトル化不可の依存関係が存在すると仮定する。」が出力される．本事例では，コンパイラが依存関係を解析できるようにベクトル化対象の DO ループを分割し，コンパイラによるベクトル化と最適化を促進する．図 6.1.6-2 にチューニング前の FTRACE 情報を記載する．

```
!チューニング前
 +------>          do k=ks,ke,kd
 |+----->           do j=js,je,jd
 ||V---->            do i=is,ie,id
 |||                   var1=zero
 |||                   var2=zero
 |||     A             var3=a(i,j,k)
 |||     A             var4=b(i,j,k)
 |||                   var5=1.0d0/var3
  :
 (省略)
  :
 |||     A             var1=var1+5.D-1*var4*(c(i,j,k-1)+c(i,j,k+1))
 |||             &                 *(d(k-1)-d(k))
 |||             &                 *e(k)*f(k)*var5
 |||     A             g(i,j,k)=var6
 |||
 |||    S              h(i,j,k)=h(i,j,k)+var1                    <-- ベクトル化の依存関係が解析できずスカラ処理
 |||    S  C
 |||    AS             if (i(i,j,k).ne.0) goto 510
 |||    S  C
 |||    S              if (i(i+1,j,k).ne.3) goto 520
 |||    S  C
 |||    S              j(i,j,k)=zero
 |||    S              var1=-k(i,j,k,var7)*5.D-1*l(i,j,k)
 |||    S       &          -5.D-1*(1.D0-k(i,j,k,var7))*(l(i,j,k)-dabs(l(i,j,k)))
 |||    S       &          +var4*m(i)**2*var5*4.0d0
 |||    S              var2=-k(i,j,k,var7)*5.D-1*l(i,j,k)
 |||    S       &          -5.D-1*(1.D0-k(i,j,k,var7))*(l(i,j,k)+dabs(l(i,j,k)))
 |||    S       &          -var4*m(i)**2*var5*4.0d0
 |||    S              h(i,j,k)=h(i,j,k)+var1*c(i+1,j,k)
 |||    S       &                        +dmax1(var2,zero)*n(i,j,k)
 |||    S              o(i,j,k)=o(i,j,k)-dmax1(-var2,zero)
 |||    S  C
 |||    S    520       if (i(i-1,j,k).ne.3) goto 530
  :
 (省略)
  :
 |||    S    510       if (i(i,j,k).ne.3) goto 500
 |||    S  C
 |||    S              h(i,j,k)=var8*n(i,j,k)
 |||    S              o(i,j,k)=-var8
 |||       C
 |||         500       continue
 |||       C
 |||    S              p(i,j,k)=h(i,j,k)
 |||    S              q(i,j,k)=o(i,j,k)
 ||V----               end do
 |+-----              end do
 +------             end do
```

図 6.1.6-1 ループ分割前のコード

```
FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS V.OP  AVER.    VECTOR I-CACHE O-CACHE   BANK CONFLICT    ADB HIT  PROC.NAME
           TIME[sec](  %)    [msec]                   RATIO V.LEN      TIME    MISS    MISS CPU PORT  NETWORK   ELEM.%

       432  331.491(  5.6)   767.339  12477.8  4938.0 93.69 256.0     56.045   0.001  45.160    0.750    2.489    65.92【チューニング前】
```

図 6.1.6-2 ループ分割前の FTRACE 情報

(2) チューニング内容

図 6.1.6-3 にチューニング後のコードを記載する. ここでは, インデックス j の DO ループを分割する. 分割した前半の DO ループで求めた変数 var1 の計算結果を後半の DO ループで参照するため, 前半の DO ループで var1 の値を作業配列に格納する. 後半の DO ループでは作業配列から var1 の値を参照する. ループ分割を行ったことにより, コンパイラは前半と後半の DO ループのベクトル化と最適化を行っている.

```
!チューニング後
 +------>          do k=ks,ke,kd
 |+----->           do j=js,je,jd
 ||V---->            DO i=is,ie,id
 |||                   var1=zero
 |||                   var2=zero
 |||     A             var3=a(i,j,k)
 |||     A             var4=b(i,j,k)
 |||                   var5=1.0d0/var3
  :
 (省略)
  :
 |||     A               var1=var1+5.D-1*var4*(c(i,j,k-1)+c(i,j,k+1))
 |||               &                 *(d(k-1)-d(k))
 |||               &                 *e(k)*f(k)*var5
 |||     A               g(i,j,k)=var6
 |||
 |||     A             wk1(i,j)=var1   <-- 作業配列に計算結果を保存
 ||V----               end do
 |+-----               end do          <-- ループを分割
 |+----->               do j=js,je,jd
 ||V---->               do i=is,ie,id
 |||     A             var1=wk1(i,j)   <-- 作業配列から参照
 |||                   var2=zero
 |||                   var3=a(i,j,k)
 |||     A             var4=b(i,j,k)
 |||     A             var5=1.0d0/var3
 |||
 |||     A             h(i,j,k)=h(i,j,k)+var1
 |||
 |||     A             if (i(i,j,k).ne.0) goto 510
 |||
 |||     A             if (i(i+1,j,k).ne.3) goto 520
 |||
 |||     A             j(i,j,k)=zero
 |||     A             var1=-k(i,j,k,var7)*5.D-1*l(i,j,k)
 |||               &       -5.D-1*(1.D0-k(i,j,k,var7))*(l(i,j,k)-dabs(l(i,j,k)))
 |||               &       +var4*m(i)**2*var5*4.0d0
 |||                   var2=-k(i,j,k,var7)*5.D-1*l(i,j,k)
 |||               &       -5.D-1*(1.D0-k(i,j,k,var7))*(l(i,j,k)+dabs(l(i,j,k)))
 |||               &       -var4*m(i)**2*var5*4.0d0
 |||     A             h(i,j,k)=h(i,j,k)+var1*n(i+1,j,k)
 |||               &                       +dmax1(var2,zero)*c(i,j,k)
 |||     A             o(i,j,k)=o(i,j,k)-dmax1(-var2,zero)
 |||
 |||     A     520     if (i(i-1,j,k).ne.3) goto 530
  :
```

```
(省略)
 :
|||     A     510    if (i(i,j,k).ne.3) goto 500
|||
|||     A            h(i,j,k)=var8*n(i,j,k)
|||     A            o(i,j,k)=-var8
|||
|||           500    continue
|||
|||     A            p(i,j,k)=h(i,j,k)
|||     A            q(i,j,k)=o(i,j,k)
||V----              end do
|+-----             end do
+------            end do
```

図 6.1.6-3 ループ分割後のコード

(3) 性能分析

図 6.1.6-4 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
432	331.491(5.6)	767.339	12477.8	4938.0	**93.69**	256.0	56.045	0.001	45.160	0.750	2.489	65.92	【チューニング前】
432	105.136(2.2)	243.370	64816.0	27014.0	**99.71**	256.0	105.100	0.001	0.012	0.000	16.167	63.73	【チューニング後】

図 6.1.6-4 ループ分割前後の FTRACE 情報

DO ループ全体をベクトル化することにより, ベクトル演算率が 93.7% から 99.7% に向上し, 実行時間が 331.5 秒から 105.1 秒に短縮されている.

6.1.7. IF 文のループ外への移動による高速化

(1) チューニング方針

DO ループ内に IF 文がある場合, その IF 文の真偽がその DO ループ実行中に不変であっても, 毎回判定処理を行う. また, SX-ACE では, DO ループがベクトル化されている場合, ベクトル処理を効率良く実行するために IF 文の真と偽の両方の処理を実行する. そのため IF 文の判定のオーバーヘッドと IF 文の偽の処理を行うために処理量が増加する. 本事例では, ベクトル化された DO ループ内で IF 文の真偽が不変である場合の高速化を示す.

図 6.1.7-1 にチューニング前のコードを記載する. ベクトル化された DO ループ内に IF 文が複数あり, それぞれの判定処理で用いている変数 var1, var2, var3, var4 は DO ループ内で不変である. IF 文をベクトル化された DO ループの外に出し, IF 文による処理の削減を図る. 図 6.1.7-2 にチューニング前の FTRACE 情報を記載する.

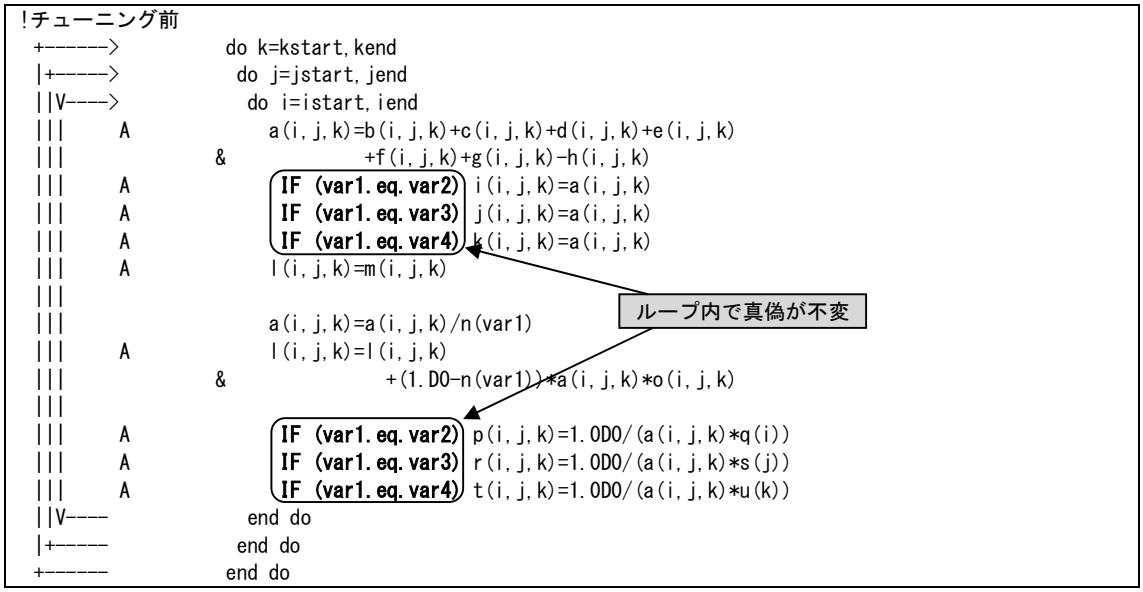


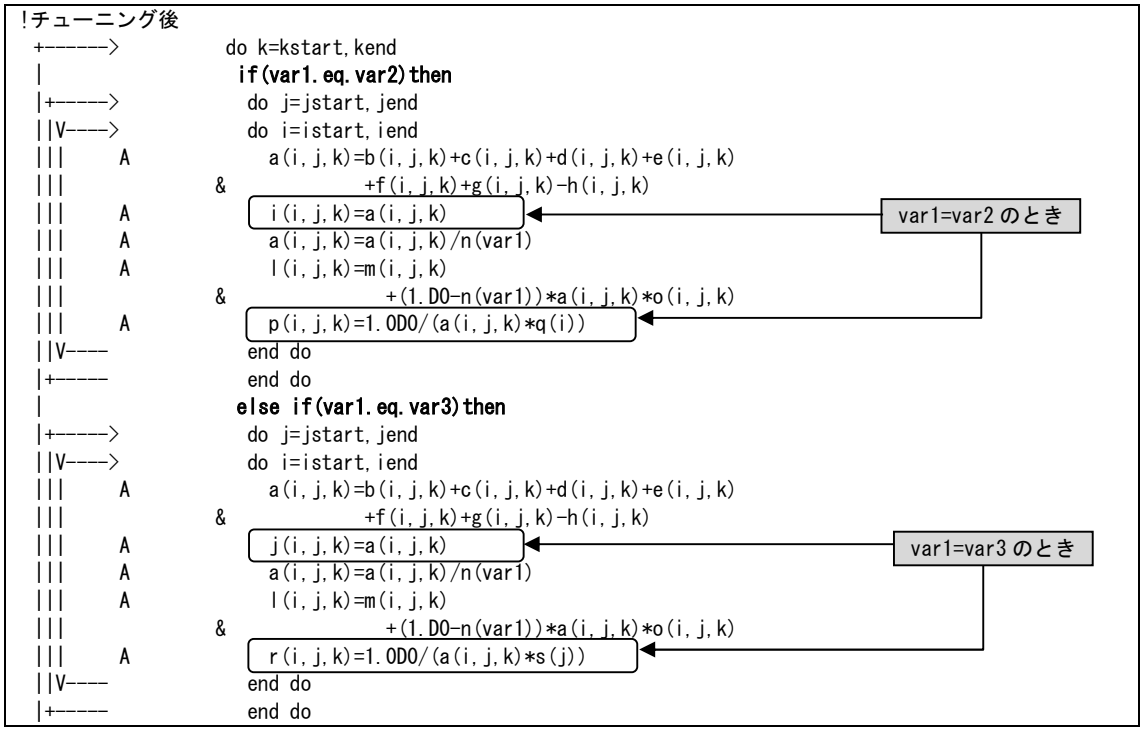
図 6.1.7-1 IF 文のループ外への移動前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
3888	121.205(2.0)	31.174	16591.0	7008.8	98.99	256.0	121.201	0.001	0.001	0.945	83.986	3.13	【チューニング前】

図 6.1.7-2 IF 文のループ外への移動前の FTRACE 情報

(2) チューニング内容

図 6.1.7-3 にチューニング後のコードを記載する. それぞれの IF 文を DO ループの外に移動し, 各条件下における演算処理に分割する.



```
|                  else if(var1.eq.var4)then
|+------>           do j=jstart,jend
||V---->           do i=istart,iend
|||     A           a(i,j,k)=b(i,j,k)+c(i,j,k)+d(i,j,k)+e(i,j,k)
|||               &           +f(i,j,k)+g(i,j,k)-h(i,j,k)
|||     A           k(i,j,k)=a(i,j,k)
|||     A           a(i,j,k)=a(i,j,k)/n(var1)
|||     A           l(i,j,k)=m(i,j,k)
|||               &             +(1.0D0-n(var1))*a(i,j,k)*o(i,j,k)
|||     A           t(i,j,k)=1.0D0/(a(i,j,k)*u(k))
||V----             end do
|+-----             end do
|                  else
|+-----> do j=jstart,jend
||V----> do i=istart,iend
|||     A           a(i,j,k)=b(i,j,k)+c(i,j,k)+d(i,j,k)+e(i,j,k)
|||               &           +f(i,j,k)+g(i,j,k)-h(i,j,k)
|||     A           a(i,j,k)=a(i,j,k)/n(var1)
|||     A           l(i,j,k)=m(i,j,k)
|||               &             +(1.0D0-n(var1))*a(i,j,k)*o(i,j,k)
||V----             end do
|+-----             end do
|                  end if
+------           end do
```

var1=var4 のとき

図 6.1.7-3 IF 文のループ外への移動後のコード

(3) 性能分析

図 6.1.7-4 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CPU PORT	CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
3888	121.205(2.0)	31.174	16591.0	7008.8	98.99	256.0	121.201	0.001	0.001	0.945	83.986	3.13	【チューニング前】
3888	39.874(0.8)	10.256	45596.6	21302.3	99.52	256.0	39.869	0.002	0.001	1.278	10.859	3.13	【チューニング後】

図 6.1.7-4 IF 文のループ外への移動前後の FTRACE 情報

IF 文をベクトル化された DO ループの外に出すことにより, IF 文の判定処理のオーバーヘッドを削減し, また, IF 文の偽の処理が削減されたことで, 実行時間が 121.2 秒から 39.9 秒に短縮されている.

6.1.8. RedBlack 法のメモリアクセス方法変更による高速化

(1) チューニング方針

本事例は RedBlack (Multi-Coloring) 法を用いた SOR 法の実装において, メモリアクセスの方法を間接参照からマスク制御による連続アクセスに修正することにより, メモリアクセスの時間を短縮させる例である.

図 6.1.8-1 にチューニング前のコード, 図 6.1.8-2 に配列データを参照するインデックスの格納処理を記載する. RedBlack 法は 1 要素飛びで計算を行うため, 1 要素飛びの配列インデックスを格納した配列 b の値を参照し, リストベクトルによる対象要素の計算を行っている. 本事例では, メモリアクセスの方法を連続アクセスに修正することでメモリアクセスの時間を短縮することができる. 図 6.1.8-3 にチューニング前の FTRACE 情報を記載する.

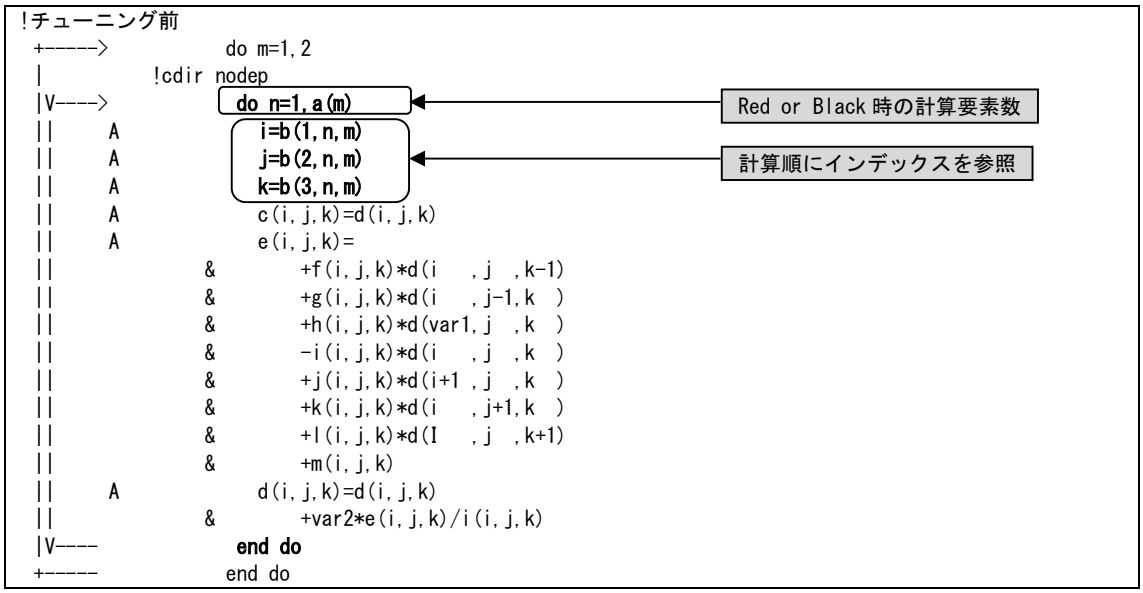


図 6.1.8-1 RedBlack 法のメモリアクセス方法変更前のコード

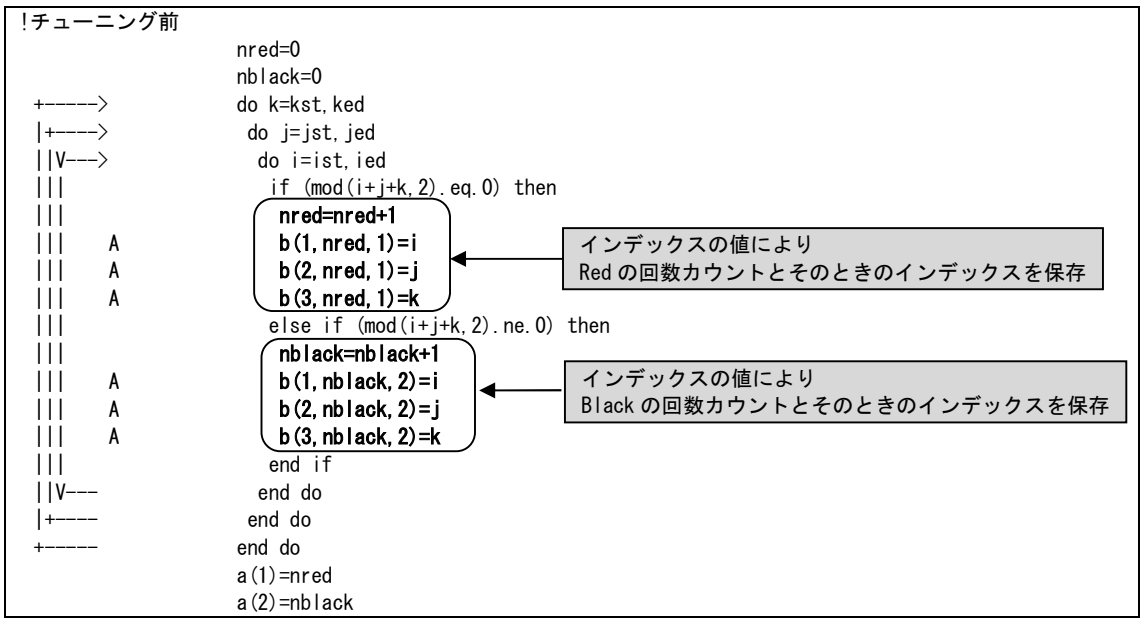


図 6.1.8-2 RedBlack 法の配列インデックス格納処理のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CPU PORT	CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
3888	121.205(2.0)	31.174	16591.0	7008.8	98.99	256.0	121.201	0.001	0.001	0.945	83.986	3.13	【チューニング前】

図 6.1.8-3 RedBlack 法のメモリアクセス方法変更前の FTRACE 情報

(2) チューニング内容

図 6.1.8-4 にチューニング後のコード, 図 6.1.8-5 にマスク制御用の配列作成のコードを記載する. 計算領域の全てを連続に参照するよう三重 DO ループの形に修正し, マスク制御用の配列を用いて計算領域を制限する. この修正により, DO ループ内で参照する配列データは連続なメモリアクセスとなるため修正前のリストベクトルに比べてメモリアクセスの時間が短縮される.

なお, SX-ACE はベクトルループ内に IF 文による真偽の判定処理がある場合, 真と偽の両方

の処理を行い，最終的に真偽による結果保持の判定を行う．そのためマスク制御用の IF 文を追加すると総演算量が 2 倍に増える．

```
!チューニング後
 +-----> do m=1,2
 |+----> do k=kst,ked
 ||+---> do j=jst,jed
 |||   !cdir nodep
 |||V--> do i=ist,ied
 ||||  A   if(itbl(i,j,k,m).ne.1) cycle
 ||||  A   c(i,j,k)=d(i,j,k)
 ||||  A   e(i,j,k)=
 ||||      &   +f(i,j,k)*d(i  ,j  ,k-1)
 ||||      &   +g(i,j,k)*d(i  ,j-1,k  )
 ||||      &   +h(i,j,k)*d(var1,j  ,k  )
 ||||      &   -i(i,j,k)*d(i  ,j  ,k  )
 ||||      &   +j(i,j,k)*d(i+1,j  ,k  )
 ||||      &   +k(i,j,k)*d(i  ,j+1,k  )
 ||||      &   +l(i,j,k)*d(I  ,j  ,k+1)
 ||||      &   +m(i,j,k)
 ||||  A   d(i,j,k)=d(i,j,k)
 ||||      &   +var2*e(i,j,k)/i(i,j,k)
 |||V--    end do
 ||+---    end do
 |+----    end do
 +-----   end do
```

全メモリ領域を順番に参照

計算箇所を制御

図 6.1.8-4 RedBlack 法のメモリアクセス方法変更後のコード

```
!チューニング後
W***=== A  itbl=0
+------>   do k=kst,ked
|+------>  do j=jst,jed
||V---->   DO i=ist,ied
|||          if (mod(i+j+k,2).eq.0) then
|||    A       itbl(i,j,k,1) = 1
|||          else
|||    A       itbl(i,j,k,2) = 1
|||          end if
||V---- A   end do
|+-----     end do
+------     end do
```

インデックスの値により Red の計算を許可

インデックスの値により Black の計算を許可

図 6.1.8-5 RedBlack 法のマスク制御用配列作成のコード

(3) 性能分析

図 6.1.8-6 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CPU PORT	CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
21027	**1758.064**(29.5)	83.610	33798.2	**4164.1**	99.97	256.0	1758.055	0.003	0.004	32.030	73.856	30.49	【チューニング前】
21027	**1059.943**(22.1)	50.409	42305.3	**13813.5**	99.52	256.0	1059.933	0.003	0.003	0.928	259.042	25.92	【チューニング後】

図 6.1.8-6 RedBlack 法のメモリアクセス方法変更前後の FTRACE 情報

マスク制御用の IF 文を追加したことにより総演算量が約 2 倍に増えているが，メモリアクセス性能の向上により実行時間は 1758.1 秒から 1059.9 秒に短縮されている．

6.1.9. ハイパープレーン法のアクセスパターン変更による高速化

(1) チューニング方針

ステンシル計算など多重 DO ループの各ループ(次元方向)に依存関係があるループをベクトル化する手法に, ハイパープレーン(超平面) 法がある. 従来のベクトルコンピュータではベクトル長の拡大を優先してすべての次元を使ってハイパープレーンを作る最適化が行われてきた. 本事例では, ハイパープレーンを作る次元を落とし, ADB を使用してメモリアクセスの効率化を図る例を示す(ADB については 5.2.2 を参照).

図 6.1.9-1 にチューニング前のコードを記載する. 抜粋した箇所では i 方向の依存関係が認められる. 配列 c (i, j, k) を計算するためには, 配列 c (i-1, j, k) の計算結果が必要となる. この依存関係が, i, j, k 全ての方向に存在するコードである.

そこで, コンパイラが依存関係を解消できるように, DO ループの 10, 20, 30 番でハイパープレーン法を用いている. 図 6.1.9-2 にチューニング前の FTRACE 情報を記載する.

```
!チューニング前
 |                    mmin=6+(ii1-1)+(jj1-1)+(kk1-1)
 |                    mmax=iie+jje+kke
 |
 |+------>            do 10 m=mmin,mmax
 ||                     kdiv1=m-(jje+iie)
 ||                     kdiv2=m-(jj2+ii2)
 ||                     ksta=max0(kdiv1,kk2)
 ||                     kend=min0(kdiv2,kke)
 ||
 ||+---->              do 20 k=ksta,kend
 |||                      idiv1=m-(k+jje)
 |||                      idiv2=m-(k+jj2)
 |||                      ista=max0(idiv1,ii2)
 |||                      iend=min0(idiv2,iie)
 |||
 |||V--->                do 30 i=ista,iend
 ||||                       j =m-k-i
 ||||                       im=i-1
 ||||                       jm=j-1
 ||||                       km=k-1
 ||||
 ||||                       var1=0.D0
 ||||                       var2=0.D0
 ||||
 ||||                       ii=im
 ||||                       jj=j
 ||||                       kk=k
 ||||                       kl=1
 ||||                       var3=1.0D0
 ||||
 ||||    A                  var4=a(ii,jj,kk,1)*b(ii,jj,kk,kl)
 ||||            &              +a(ii,jj,kk,2)*b(ii,jj,Kk,kl+3)
 ||||            &              +a(ii,jj,kk,3)*b(ii,jj,kk,kl+6)
 ||||    A                  var5=a(i,j,k,1)*b(i,j,k,kl)
 ||||            &              +a(i,j,k,2)*b(i,j,k,kl+3)
 ||||            &              +a(i,j,k,3)*b(i,j,k,kl+6)
 ||||                       var6=b(ii,jj,kk,kl)*b(ii,jj,kk,kl)
 ||||            &              +b(ii,jj,kk,kl+3)*b(ii,jj,kk,kl+3)
 ||||            &              +b(ii,jj,kk,kl+6)*b(ii,jj,kk,kl+6)
 ||||                       var7=b(i,j,k,kl)*b(i,j,k,kl)
 ||||            &              +b(i,j,k,kl+3)*b(i,j,k,kl+3)
 ||||            &              +b(i,j,k,kl+6)*b(i,j,k,kl+6)
```

```
 :
(省略)
 :
||||    A            var8=c(ii,jj,kk,1)/d(ii,jj,kk)
||||    A            var9=c(ii,jj,kk,2)/d(ii,jj,kk)
||||    A            var10=c(ii,jj,kk,3)/d(ii,jj,kk)
||||    A            var11=c(ii,jj,kk,4)/d(ii,jj,kk)
||||    A            var12=c(ii,jj,kk,5)/d(ii,jj,kk)
||||    A            var13=c(ii,jj,kk,6)/d(ii,jj,kk)
||||    A            var14=c(ii,jj,kk,7)/d(ii,jj,kk)
||||    A            var15=c(ii,jj,kk,8)/d(ii,jj,kk)
||||
||||    A            var16=c(ii,jj,kk,9)/d(ii,jj,kk)
||||    A            var17=c(ii,jj,kk,10)/d(ii,jj,kk)
 :
(省略)
 :
||||    A          c(i,j,k,1)=var18*(var19*e(i,j,k,1)+f(i,j,k)
||||              & *var20*(var21 + var22 + var23))
||||    A          c(i,j,k,2)=var18*(var19*e(i,j,k,2)+f(i,j,k)
||||              & *var20*(var24 + var25 + var26))
||||    A          c(i,j,k,3)=var18*(var19*e(i,j,k,3)+f(i,j,k)
||||              & *var20*(var27 + var28 + var29))
||||    A          c(i,j,k,4)=var18*(var19*e(i,j,k,4)+f(i,j,k)
||||              & *var20*(var30 + var31 + var32))
||||    A          c(i,j,k,5)=var18*(var19*e(i,j,k,5)+f(i,j,k)
||||              & *var20*(var33 + var34 + var35))
||||    A          c(i,j,k,6)=var18*(var19*e(i,j,k,6)+f(i,j,k)
||||              & *var20*(var36 + var37 + var38))
||||    A          c(i,j,k,7)=var18*(var19*e(i,j,k,7)+f(i,j,k)
||||              & *var20*(var39 + var40 + var41))
||||    A          c(i,j,k,8)=var18*(var19*e(i,j,k,8)+f(i,j,k)
||||              & *var20*(var42 + var43 + var44))
||||
||||    A          c(i,j,k,9)=var48*(var19*e(i,j,k,9)+f(i,j,k)
||||              & *var20*(var45 + var46 + var47))
||||    A          c(i,j,k,10)=var48*(var19*e(i,j,k,10)+f(i,j,k)
||||              & *var20*(var49 + var50 + var 51))
||||
|||V---        30       continue
||+----        20     continue
|+-----        10   continue
```

図 6.1.9-1 ハイパープレーン法のアクセスパターン変更前のコード

```
FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS V.OP  AVER.     VECTOR I-CACHE O-CACHE   BANK CONFLICT    ADB HIT  PROC.NAME
           TIME[sec](  %  )    [msec]                    RATIO V.LEN      TIME    MISS    MISS CPU PORT  NETWORK   ELEM.%

      500  200.511( 20.7)   401.023   9068.3   3825.1 97.51  37.7   198.372   0.003   0.012    3.954  122.914    70.49【チューニング前】
```

図 6.1.9-2 ハイパープレーン法のアクセスパターン変更前の FTRACE 情報

(2) チューニング内容

図 6.1.9-3 にてチューニング前後のアクセスパターンの違いを示す. ここで同じ色の格子が同一のハイパープレーンになっている. 左側がチューニング前のハイパープレーンで i, j, k で作る 3 次元空間にまたがった 2 次元平面になっており, 右側がチューニング後のハイパープレーンで i, j で作る平面上の直線になっている. それぞれ以下のように処理が行われる.

「黄色のハイパープレーンの処理」→「赤色のハイパープレーンの処理」→「青色のハイパープレーンの処理」→・・・→「黒色のハイパープレーンの処理」→「灰色のハイパープレーンの処理」

チューニング前は 3 次元に跨る処理になっており, メモリ上で離れた位置のデータを参照する

ことになる. チューニング後は 2 次元上でハイパープレーンを作り, メモリ上で近い位置のデータを参照している. そのため, ブロックロードや ADB によるデータの再利用性が高くなっている.

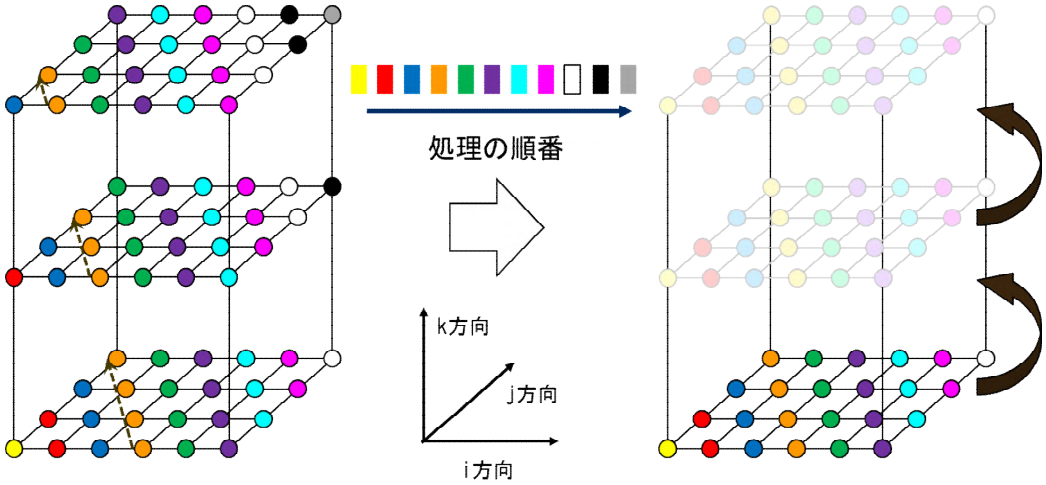


図 6.1.9-3 ハイパープレーン法のアクセスパターン変更イメージ

図 6.1.9-4 にチューニング後のコードを記載する. 計算内容に変更はなく, DO ループの制御のみでアクセスパターンの変更を行っている. DO ループの 10 番は k 方向の開始から終了まで通常の DO ループで構成される. 残りの DO ループの 20, 30 番によって, i, j の 2 次元でハイパープレーンを作っている.

```
!チューニング後
 |                  mmin=4+(ii1-1)+(jj1-1)
 |                  mmax=iie+jje
 |
 |+------>          do 10 k=kk2,kke          ← 通常の DO ループ
 ||
 ||+---->           do 20 m=mnix,mmax
 |||                  idiv1=m-jje
 |||                  idev2=m-jj2
 |||                  ista=max0(idve1,ii2)   ← 2 次元のハイパープレーン法を適用
 |||                  iend=min0(idiv2,ii3)
 |||
 |||V--->           do 30 i=istart,iend
 ||||                 j =m-I
 ||||                 im=i-1
 ||||                 jm=j-1
 ||||                 km=k-1
 ||||
 ||||                 var1=0.DO
 ||||                 var2=0.DO
 ||||
 ||||                 ii=im
 ||||                 jj=j
 ||||                 kk=k
 ||||                 kl=1
 ||||                 var3=1.0DO
 ||||
 ||||    A            var4=a(ii,jj,kk,1)*b(ii,jj,kk,kl)
 ||||          &           +a(ii,jj,kk,2)*b(ii,jj,kk,kl+3)
 ||||          &           +a(ii,jj,kk,3)*b(ii,jj,kk,kl+6)
```

```
 ||||    A                 var5=a(i,j,k,1)*b(i,j,k,kl)
 ||||              &           +a(i,j,k,2)*b(i,j,k,kl+3)
 ||||              &           +a(i,j,k,3)*b(i,j,k,kl+6)
 ||||                        var6=b(ii,jj,kk,kl)*b(ii,jj,kk,kl)
 ||||              &           +b(ii,jj,kk,kl+3)*b(ii,jj,kk,kl+3)
 ||||              &           +b(ii,jj,kk,kl+6)*b(ii,jj,kk,kl+6)
 ||||                        var7=b(i,j,k,kl)*b(i,j,k,kl)
 ||||              &           +b(i,j,k,kl+3)*b(i,j,k,kl+3)
 ||||              &           +b(i,j,k,kl+6)*b(i,j,k,kl+6)
  :
(省略)
  :
 ||||    A           var8=c(ii,jj,kk,1)/d(ii,jj,kk)
 ||||    A           var9=c(ii,jj,kk,2)/d(ii,jj,kk)
 ||||    A           var10=c(ii,jj,kk,3)/d(ii,jj,kk)
 ||||    A           var11=c(ii,jj,kk,4)/d(ii,jj,kk)
 ||||    A           var12=c(ii,jj,kk,5)/d(ii,jj,kk)
 ||||    A           var13=c(ii,jj,kk,6)/d(ii,jj,kk)
 ||||    A           var14=c(ii,jj,kk,7)/d(ii,jj,kk)
 ||||    A           var15=c(ii,jj,kk,8)/d(ii,jj,kk)
 ||||
 ||||    A           var16=c(ii,jj,kk,9)/d(ii,jj,kk)
 ||||    A           var17=c(ii,jj,kk,10)/d(ii,jj,kk)
  :
(省略)
  :
 ||||    A         c(i,j,k,1)=var18*(var19*e(i,j,k,1)+f(i,j,k)
 ||||              & *var20*(var21 + var22 + var23))
 ||||    A         c(i,j,k,2)=var18*(var19*e(i,j,k,2)+f(i,j,k)
 ||||              & *var20*(var24 + var25 + var26))
 ||||    A         c(i,j,k,3)=var18*(var19*e(i,j,k,3)+f(i,j,k)
 ||||              & *var20*(var27 + var28 + var29))
 ||||    A         c(i,j,k,4)=var18*(var19*e(i,j,k,4)+f(i,j,k)
 ||||              & *var20*(var30 + var31 + var32))
 ||||    A         c(i,j,k,5)=var18*(var19*e(i,j,k,5)+f(i,j,k)
 ||||              & *var20*(var33 + var34 + var35))
 ||||    A         c(i,j,k,6)=var18*(var19*e(i,j,k,6)+f(i,j,k)
 ||||              & *var20*(var36 + var37 + var38))
 ||||    A         c(i,j,k,7)=var18*(var19*e(i,j,k,7)+f(i,j,k)
 ||||              & *var20*(var39 + var40 + var41))
 ||||    A         c(i,j,k,8)=var18*(var19*e(i,j,k,8)+f(i,j,k)
 ||||              & *var20*(var42 + var43 + var44))
 ||||
 ||||    A         c(i,j,k,9)=var48*(var19*e(i,j,k,9)+f(i,j,k)
 ||||              & *var20*(var45 + var46 + var47))
 ||||    A         c(i,j,k,10)=var48*(var19*e(i,j,k,10)+f(i,j,k)
 ||||              & *var20*(var49 + var50 + var51))
 ||||
 |||V---        30       continue
 ||+----        20     continue
 |+-----        10   continue
```

図 6.1.9-4 ハイパープレーン法のアクセスパターン変更後のコード

(3) 性能分析

図 6.1.9-5 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CPU PORT	CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
500	200.511(20.7)	401.023	9068.3	3825.1	97.51	37.7	198.372	0.003	0.012	3.954	122.914	**70.49**	【チューニング前】
500	138.447(16.3)	276.895	13018.7	5539.9	97.92	37.7	135.649	0.002	2.373	3.469	61.125	**96.56**	【チューニング後】

図 6.1.9-5 ハイパープレーン法のアクセスパターン変更前後の FTRACE 情報

メモリアクセスを効率化することにより，ADB ヒット率が 70.49% から 96.56% に向上し，実行時間が 200.5 秒から 138.4 秒に短縮されている．

6.1.10. 作業配列の変数化によるメモリアクセス負荷低減による高速化

(1) チューニング方針

計算処理において一時的に作業配列を使ってデータを保持させることがある．しかし，作業配列の容量が大きい場合，そのデータをメモリにストアする時間が長くなり，性能が低下することがある．本事例では，作業配列を作業変数に置き換え，不要な配列データのストア処理を削減し，高速化を図る．

図 6.1.10-1 にチューニング前のコードを記載する．DO ループ内の冒頭で計算結果を作業配列 wk1， wk2 に格納し，それ以降の処理では配列に格納した値を参照する．しかし，当該ループ以降ではその作業配列は参照されない値である．計算結果の参照を配列から変数へ変更することにより，メモリへのデータ書き込み処理を削減することができる．図 6.1.10-2 にチューニング前の FTRACE 情報を記載する．

```
!チューニング前
 +------>           do k=ks,ke,kd
 |+----->            do j=js,je,jd
 ||V---->             do i=is,ie,id
 |||     A              wk2(i,j,k)=5.D-1*(a(i,j-1,k)+a(i+1,j-1,k))
 |||     A              wk1(i,j,k)=5.D-1*(a(i,j,k)+a(i+1,j,k))
  :
 (省略)
  :
 |||     A              b(i,j,k)=zero
 |||     A              var1=-c(i,j,k,var2)*5.D-1*wk1(i,j,k)
 |||                &         -5.D-1*(1.D0-c(i,j,k,var2))
 |||                &               *(wk1(i,j,k)+dabs(wk1(i,j,k)))
 |||                &         +d(i,j,k)*e(j)*var3*var4/(2.5D-1*f(j)*f(j))
 |||                    var5=-c(i,j,k,var2)*5.D-1*wk1(i,j,k)
 |||                &         -5.D-1*(1.D0-c(i,j,k,var2))
 |||                &               *(wk1(i,j,k)+dabs(wk1(i,j,k)))
 |||                &         -d(i,j,k)*e(j)*var3*var4/(2.5D-1*f(j)*f(j))
 |||     A              g(i,j,k)=g(i,j,k)+var1*h(i,j+1,k)
 |||                &                         +dmax1(var5,zero)*h(i,j,k)
 |||     A              i(i,j,k)=i(i,j,k)-dmax1(-var5,zero)
 |||
 |||     A     130      if(j(i,j-1,k).ne.3) goto 140
 |||     A              k(i,j,k)=zero
 |||     A              var1=c(i,j,k,var2)*5.D-1*wk2(i,j,k)
 |||                &         +5.D-1*(1.D0-c(i,j,k,var2))
 |||                &               *(wk2(i,j,k)+dabs(wk2(i,j,k)))
 |||                &         +d(i,j,k)*e(j)*var3*var4/(2.5D-1*f(j)*f(j))
 |||                    var5=c(i,j,k,var2)*5.D-1*wk2(i,j,k)
 |||                &         +5.D-1*(1.D0-c(i,j,k,var2))
 |||                &               *(wk2(i,j,k)+dabs(wk2(i,j,k)))
 |||                &         -d(i,j,k)*e(j)*var3*var4/(2.5D-1*f(j)*f(j))
 |||     A              g(i,j,k)=g(i,j,k)+var1*h(i,j-1,k)
 |||                &                         +dmax1(var5,zero)*h(i,j,k)
 |||     A              i(i,j,k)=i(i,j,k)-dmax1(-var5, zero)
 |||
 |||     A     140      if (j(i,j,k+1).ne.3) goto 150
  :
 (省略)
```

```
  :
|||     A     110     if(j(i,j,k).ne.3) goto 100
|||     A             g(i,j,k)=vsr6*h(i,j,k)
|||     A             i(i,j,k)=-var6
|||           100     continue
||V----              end do
|+-----             end do
+------            end do
```

図 6.1.10-1 作業配列の変数化前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CPU PORT	CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
432	111.234(1.9)	257.486	40862.9	17260.1	99.85	256.0	111.233	0.000	0.001	0.613	24.591	51.70	【チューニング前】

図 6.1.10-2 作業配列の変数化前の FTRACE 情報

(2) チューニング内容

図 6.1.10-3 にチューニング後のコードを記載する. 計算結果の参照を作業配列から作業変数に変更することによりメモリアクセス回数を削減することができる.

```
!チューニング後
 +------>          do k=ks,ke,kd
 |+----->           do j=js,je,jd
 ||V---->            do i=is,ie,id
  :
(省略)
  :
|||     A             b(i,j,k)=zero
|||     A             wk_var1=5.D-1*(h(i,j,k)+h(i+1,j,k))
|||     A             var1=-c(i,j,k,var2)*5.D-1*wk_var1
|||               &         -5.D-1*(1.D0-c(i,j,k,var2))
|||               &               *(wk_var1+dabs(wk_var1))
|||               &         +d(i,j,k)*e(j)*var3*var4/(2.5D-1*f(j)*f(j))
|||                   var5=-c(i,j,k,var2)*5.D-1*wk_var1
|||               &         -5.D-1*(1.D0-c(i,j,k,var2))
|||               &               *(wk_var1+dabs(wk_var1))
|||               &         -d(i,j,k)*e(j)*var3*var4/(2.5D-1*f(j)*f(j))
|||     A             g(i,j,k)=g(i,j,k)+var1*h(i,j+1,k)
|||               &                       +dmax1(var5,zero)*h(i,j,k)
|||     A             i(i,j,k)=i(i,j,k)-dmax1(-var5,zero)
|||
|||     A     130     if(j(i,j-1,k).ne.3) goto 140
|||     A             k(i,j,k)=zero
|||     A             wk_var2=5.D-1*(h(i,j-1,k)+h(i+1,j-1,k))
|||     A             var1=c(i,j,k,var2)*5.D-1*wk_var2
|||               &       +5.D-1*(1.D0-c(i,j,k,var2))
|||               &             *(wk_var2+dabs(wk_var2))
|||               &       +d(i,j,k)*e(j)*var3*var4/(2.5D-1*f(j)*f(j))
|||                   var5=c(i,j,k,var2)*5.D-1*wk_var2
|||               &       +5.D-1*(1.D0-c(i,j,k,var2))
|||               &             *(wk_var2+dabs(wk_var2))
|||               &       -d(i,j,k)*e(j)*var3*var4/(2.5D-1*f(j)*f(j))
|||     A             g(i,j,k)=g(i,j,k)+var1*h(i,j-1,k)
|||               &                       +dmax1(var5,zero)*h(i,j,k)
|||     A             i(i,j,k)=i(i,j,k)-dmax1(-var5,zero)
|||
|||     A     140     if (j(i,j,k+1).ne.3) goto 150
  :
(省略)
  :
|||     A     110     if(j(i,j,k).ne.3) goto 100
```

作業変数に格納
作業変数を参照
作業変数を参照
作業変数に格納
作業変数を参照
作業変数を参照

```
 |||      A              g(i,j,k)=var6*h(i,j,k)
 |||      A              i(i,j,k)=-var6
 |||            100      continue
 ||V----                end do
 |+-----                end do
 +------               end do
```

図 6.1.10-3 作業配列の変数化後のコード

(3) 性能分析

図 6.1.10-4 にチューニング前後の性能値を示す.

```
FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS V.OP  AVER.    VECTOR I-CACHE O-CACHE   BANK CONFLICT    ADB HIT  PROC.NAME
           TIME[sec](  %  )    [msec]                    RATIO V.LEN      TIME    MISS    MISS CPU PORT  NETWORK   ELEM.%

       432   111.234(  1.9)   257.486  40862.9  17260.1 99.85 256.0   111.233   0.000   0.001    0.613   24.591    51.70【チューニング前】
       432    94.444(  2.0)   218.621  46164.4  19579.1 99.84 256.0    94.444   0.000   0.001    0.000   13.617    63.12【チューニング後】
```

図 6.1.10-4 作業配列の変数化前後の FTRACE 情報

メモリアクセス回数を削減することにより, 実行時間が 111.2 秒から 94.4 秒に短縮されている.

6.1.11. 命令の追い越しによる高速化

(1) チューニング方針

DO ループ内の配列アクセスにおいて, リストベクトルのロードとストアは, メモリに連続アクセスする連続ベクトルのロードとストアに比べて低速である. 本事例では, リストベクトルのロードとストアを他の処理とオーバーラップさせて, ロードとストアのレイテンシを隠ぺいし, 実行時間を短縮する例を示す.

図 6.1.11-1 のコードでは, 配列 b, c の 2 次元目の添字は, 配列 a(j) で定義されており, リストベクトルとなっている. 本 DO ループでは他にも多くの配列がリストベクトルとなっている. ループ中に現れるリストベクトルのロードとストアにおいて, メモリ上のアドレスに重なりがないことが明白な場合, 命令の実行順を変更することで, メモリアクセスのレイテンシを隠ぺいする高速化が可能である. そこで指示行の挿入による命令の追い越しを行う. 図 6.1.11-2 にチューニング前の FTRACE 情報を記載する.

```
!チューニング前
 |V----->            do 1100 j=jminq,jmax
 ||      A          var1=a(j)
  :
 ||      A          var2=b(var3,var1)
 ||      A          var4=b(var5,var1)
 ||      A          var6=c(var7,var1)
 ||      A          var8=c(var7,var1)
 ||      A          var9=c(var7,var1)
  :
 ||      A          var10= d(var11,var2)
 ||      A          var12= d(var13,var2)
 ||      A          var14= d(var15,var2)
 ||      A          var16= d(var17,var2)
  :
 ||      A           e(var18,var2) =e(var18,var2) -var19
 ||      A           e(var20,var2) =e(var20,var2) -var21+var22
 ||      A           e(var23,var2) =e(var23,var2) -var24+var25
 ||      A           e(var26,var2) =e(var26,var2) -var27+var28
```

```
 ||       A           e(var29,var2) =e(var29,I2) -var30+var31
 ||       A           f(var2)=f(var2)-var32+var33*var10
  :
 |V-----      1100    continue
```

図 6.1.11-1 命令の追い越し適用前のコード

```
FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS V.OP  AVER.     VECTOR I-CACHE O-CACHE   BANK CONFLICT    ADB HIT  PROC.NAME
           TIME[sec](  %)    [msec]                    RATIO V.LEN       TIME    MISS    MISS CPU PORT  NETWORK   ELEM.%

    10000    70.914( 7.7)      7.091  33241.6  12977.9 99.92 249.8     70.498   0.006   0.057    3.825    3.649    65.73【チューニング前】
```

図 6.1.11-2 命令の追い越し適用前の FTRACE 情報

(2) チューニング内容

図 6.1.11-3 にチューニング後のコードを記載する. GTHREORDER 指示行, VOVERTAKE 指示行を指定することで, 命令の追い越しを行いメモリアクセスのレイテンシを隠ぺいする.

```
!チューニング後
 |          !cdir gthreorder
 |          !cdir vovertake,vob        <-- 指示行の挿入
 |V----->           do 1100 j=jminq,jmax
 ||       A         var1=a(j)
  :
 ||       A         var2=b(var3,var1)
 ||       A         var4=b(var5,var1)
 ||       A         var6=c(var7,var1)
 ||       A         var8=c(var7,var1)
 ||       A         var9=c(var7,var1)
  :
 ||       A         var10= d(var11,var2)
 ||       A         var12= d(var13,var2)
 ||       A         var14= d(var15,var2)
 ||       A         var16= d(var17,var2)
  :
 ||       A          e(var18,var2) =e(var18,var2) -var19
 ||       A          e(var20,var2) =e(var20,var2) -var21+var22
 ||       A          e(var23,var2) =e(var23,var2) -var24+var25
 ||       A          e(var26,var2) =e(var26,var2) -var27+var28
 ||       A          e(var29,var2) =e(var29,I2) -var30+var31
 ||       A          f(var2)=f(var2)-var32+var33*var10
  :
 |V-----      1100    continue
```

図 6.1.11-3 命令の追い越し適用後のコード

GTHREORDER 指示行は, 直後の DO ループ中に現れるリストベクトルのロードとストアがメモリ上互いに重なることがないと仮定して, コンパイラが命令の並び換えを行う指示行である. 図 6.1.11-4 に, GTHREORDER 指示行を使用した簡単なコード例を示す. 配列 a の添字が配列 ind で定義されており, リストベクトルとなっている. GTHREORDER 指示行の指定の有無による命令発行の順番と各命令実行のタイミングチャートのイメージを図 6.1.11-5 に示す. VLD はベクトルロード命令, VGT はリストベクトルのロード命令, VSC はリストベクトルのストア命令を表す. GTHREORDER 指示行がない場合, VSC 命令(④)と VGT 命令(⑤)の依存関係が不明であるため, ④の命令を追い越して⑤の命令を実行することができず, ⑤のレイテンシが見えてしまう. ただし, ind (i) と ind (i) +1 にメモリ上の重なりがない(同じデータがない)ことが明白な場合,

GTHREORDER 指示行を指定することで，④の命令を⑤の命令が追い越すようにコンパイラは命令を並び換える．この並び換えにより，VGT 命令（③’，④’）が連続処理され，メモリアクセスのレイテンシが隠ぺいされて高速化が図れる．

```
!cdir gthreorder
do i = 1,n
 a(ind(i)) = a(ind(i)) + x(i)
 a(ind(i)+1) = a(ind(i)+1) + x(i)
enddo
```

図 6.1.11-4 GTHREORDER 指示行の使用例

図 6.1.11-5 命令発行の順番と各命令実行のタイミングチャート

VOVERTAKE 指示行は，直後の実行文中に現れる配列式またはベクトルストアを，後続のベクトルロードが追い越して実行することを許可する指示行であり，本指示行が指定された場合，プログラム実行時にハードウェアが命令の追い越しを行う．VOVERTAKE 指示行は，指示行の有効範囲が直後の DO ループではなく，VOB 指示行で指定した範囲で有効となる．VOB 指示行は，直後の実行文中に現れる DO ループ終了後，ベクトルストアの追い越しを許可しない指示行であるため，VOVERTAKE 指示行と，VOB 指示行を同時に指定することで，直後の DO ループが追い越しの有効範囲となる．図 6.1.11-6 に VOVERTAKE 指示行を使用した簡単なコード例を，図 6.1.11-7 に，VOVERTAKE 指示行を指定した場合としなかった場合の命令実行のタイミングチャートのイメージを示す．VOVERTAKE 指示行がない場合，VSC 命令（④-1）と VLD 命令（①-2，②-2），VGT 命令（③-2）のアドレスの依存関係が不明なため，VLD 命令，VGT 命令は VSC 命令を追い越して実行することができない．しかし，配列 a と，配列 ind ，x にメモリ上の重なりがなく，さらに，配列 a 自身も全要素にメモリ上の重なりがない場合には，VOVERTAKE 指示行を指定することで，④-1 の命令を，①-2，②-2，③-2 の命令が追い越すようにハードウェアが命令を実行できるようになり，メモリアクセスのレイテンシを隠ぺいすることができる．

```
!cdir vovertake, vob
do i = 1, 512
 a(ind(i)) = a(ind(i)) + x(i)
enddo
```

図 6.1.11-6 VOVERTAKE 指示行の使用例

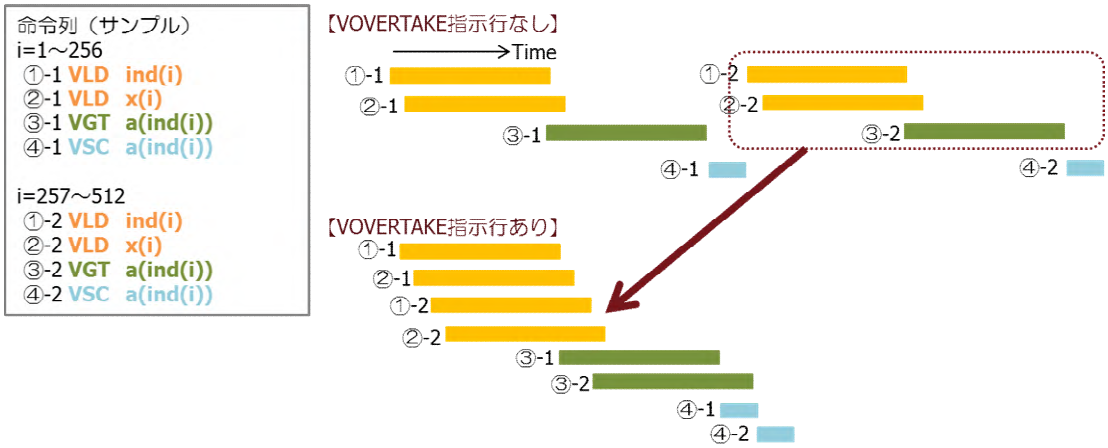


図 6.1.11-7 命令実行のタイミングチャート

GTHREORDER 指示行，VOVERTAKE 指示行が指定された場合，配列にメモリ上の重なりがあり，本来命令の並び換えができない場合であっても命令の追い越しが行われる．その場合，実行結果が不正になる場合があるため，本指示行を使用する場合には注意が必要である．

(3) 性能分析

図 6.1.11-8 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
10000	70.914(7.7)	7.091	33241.6	12977.9	99.92	249.8	70.498	0.006	0.057	3.825	3.649	65.73	【チューニング前】
10000	62.850(7.1)	6.285	37081.5	14643.0	99.92	249.8	62.430	0.005	0.059	3.860	3.853	64.56	【GTHREORDER のみ】
10000	69.735(8.5)	6.974	33803.5	13197.3	99.92	249.8	69.308	0.005	0.067	3.835	8.766	65.75	【VOVERTAKE のみ】
10000	61.827(7.0)	6.183	37695.2	14885.4	99.92	249.8	61.406	0.005	0.060	3.860	4.448	64.51	【GTHREORDRR+VOVERTAKE】

図 6.1.11-8 命令の追い越し適用前後の FTRACE 情報

GTHREORDER 指示行，VOVERTAKE 指示行を指定することで，メモリアクセスのレイテンシが隠ぺいされ，実行時間が 70.9 秒から 61.8 秒に短縮されている．

6.1.12. ADB によるメモリアクセス低減による高速化

(1) チューニング方針

5.2.2 節に記載しているように，SX-ACE では ADB を使用することでベクトル命令によるメモリアクセスを効率化することができる．本事例では，ADB によるデータの再利用性を高める最適化の例を示す．

図 6.1.12-1 にチューニング前のコードを記載する．最内ループがベクトル化されており，コンパイラはベクトル化対象の DO ループのループ変数 i でアクセスされる配列 a, b, d について ADB でバッファリングを行っている．図 6.1.12-2 にチューニング前の FTRACE 情報を記載する．ADB ヒット率 が 15.68% と低く，ADB による配列の再利用が効果的に実施できていないことがわかる．そこで，チューニング前のコードにおいて配列の再利用性に着目してみる．配列 a, d は内側の二重ループ（ループ変数 i, j の DO ループ）が実行されている間に全要素のアクセスを行うため配列の再利用性はない．一方で，配列 b は最内ループのループ変数 i に依存した 1 次元

目の要素だけにアクセスするため，ループ変数 j の DO ループが実行されるたびに，再利用することができる．ADB でバッファリングを行う配列 a, b, d の 3 つの配列のうち，内側の二重ループ実行時に再利用性があるのが配列 b のみとなっているため ADB ヒット率が低く，効率的な ADB の利用ができていない．そこで，三重 DO ループの順番を入れ替えることで配列の再利用性を高め ADB によるメモリアクセスの低減を行う．

```
!チューニング前
 +------>           do k=1,nk
 |+----->             do j=1,nj
 ||V---->A              do i=1,ni
 |||     A                a(i,j)=a(i,j)+b(i,k)*c(k,j)+d(i,j)
 ||V-----              end do
 |+-----              end do
 +------            end do
```

図 6.1.12-1 ADB によるメモリアクセス低減前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CPU PORT	CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
1	28.372(99.8)	28371.970	3914.6	1816.6	96.68	64.0	28.371	0.000	0.000	0.000	9.812	**15.68**	【チューニング前】

図 6.1.12-2 ADB によるメモリアクセス低減前の FTRACE 情報

(2) チューニング内容

図 6.1.12-3 にチューニング後のコードを記載する．最外ループと，真ん中の DO ループを入れ替え，最外ループを変数 j の DO ループとした．この場合，内側の二重ループ（ループ変数 i, k の DO ループ）が実行されている間，配列 b は全要素のアクセスを行うため再利用性はないが，配列 a, d については最内ループのループ変数 i に依存した 1 次元目の要素だけにアクセスするため，ループ変数 k の DO ループが実行されるたびに，再利用することができる．この場合，チューニング前に比べて，再利用可能な配列が増えるため ADB ヒット率が向上し，メモリアクセスを低減することができる．

```
!チューニング後
 +------>           do j=1,nj
 |+----->             do k=1,nk
 ||V---->A              do i=1,ni
 |||     A                a(i,j)=a(i,j)+b(i,k)*c(k,j)+d(i,j)
 ||V-----              end do
 |+-----              end do
 +------            end do
```

ループの入れ替え

図 6.1.12-3 ADB によるメモリアクセス低減後のコード

(3) 性能分析

図 6.1.12-4 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CPU PORT	CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
1	28.372(99.8)	28371.970	3914.6	1816.6	96.68	64.0	28.371	0.000	0.000	0.000	9.812	**15.68**	【チューニング前】
1	6.865(99.4)	6865.300	12326.5	7507.3	96.43	64.0	6.865	0.000	0.000	0.000	5.385	**34.99**	【チューニング後】

図 6.1.12-4 ADB によるメモリアクセス低減前後の FTRACE 情報

再利用可能な配列が増えたことで，ADB ヒット率が 15.68% から 34.99% に向上し，実行時間が 28.4 秒から 6.9 秒に短縮されている．

6.1.13. リストアクセスの削減による高速化

(1) チューニング方針

リストベクトルの処理は，連続ベクトルの処理に比べて時間がかかるため，本事例では可能な限りリストベクトルの処理を減らすことで高速化を図っている．図 6.1.13-1 にチューニング前のコードを記載する．ベクトル化された DO ループ内の計算処理において，配列 e はリストベクトルになっている．本コードにおいて，配列 e をリストベクトルにしているのは，単に配列要素の並び換えを行うためである．そこで，配列 e の計算処理については連続ベクトルで処理し，最後にリストベクトルを用いて配列要素の並び換えを行うことで，リストアクセスを削減し処理の高速化を行う．

```
!チューニング前
 +------>          do 20 ity=1,ntype
 |                   num=abs(a(ity))
 |+----->            do 22 k=1,num
 ||                    var1=b(k,ity)
 ||                  if ( var1.ge.c(my_rank). and.
 ||                &       var1.le.d(my_rank) ) then
 ||                  if ( k.eq.1 ) then
 ||                  e(1)=e(1)+num*f(1,ity)
 ||                  endif
 ||V---->              do 80 ig=2,ng
 |||     A             jg=g(ig)
 |||                   var2=var3*h(4,ig)
 |||     A             var4=  h(1,ig)*i(1,var1) + h(2,ig)*i(2,var1)
 |||                &         + h(3,ig)*i(3,var1)
 |||                   var4=var4*var5
 |||     A             j(ig)=dcmplx(cos(var4),-sin(var4))
 |||     A             e(jg)=e(jg)+j(ig)*f(ig,ity)
 ||V----        80     continue
 ||+---->          do 52 ia=1,k(ity)
 |||V--->          do 82 ig=2,ng
 ||||    A         jg=g(ig)
 ||||    A         var2=var3*h(4,ig)
 ||||    A         e(jg)=e(jg)+l(ity)*m(ia,ity)*var6/var2
 ||||              &                 *j(ig)*exp(-0.25D0*var2*var7)
 |||V---       82 continue
 ||+----       52 continue
 |+-----       22   continue
 +------       20 continue
```

リストベクトル

図 6.1.13-1 リストアクセス削減前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
96096	12984.005(9.5)	135.115	10777.3	5016.0	99.40	253.6	11083.951	13.631	4.866	215.473	3830.030	5.48	【チューニング前】

図 6.1.13-2 リストアクセス削減前の FTRACE 情報

(2) チューニング内容

図 6.1.13-3 に，チューニング後のコードを記載する．前処理として計算結果を格納するための作業配列 wk1 を定義する．ベクトル化された DO ループ内の計算では，計算結果を連続ベクト

ルである作業配列 wk1 に一時的に格納しておく. 計算処理終了後, 配列 e の並び換えを行う DO ループを追加している.

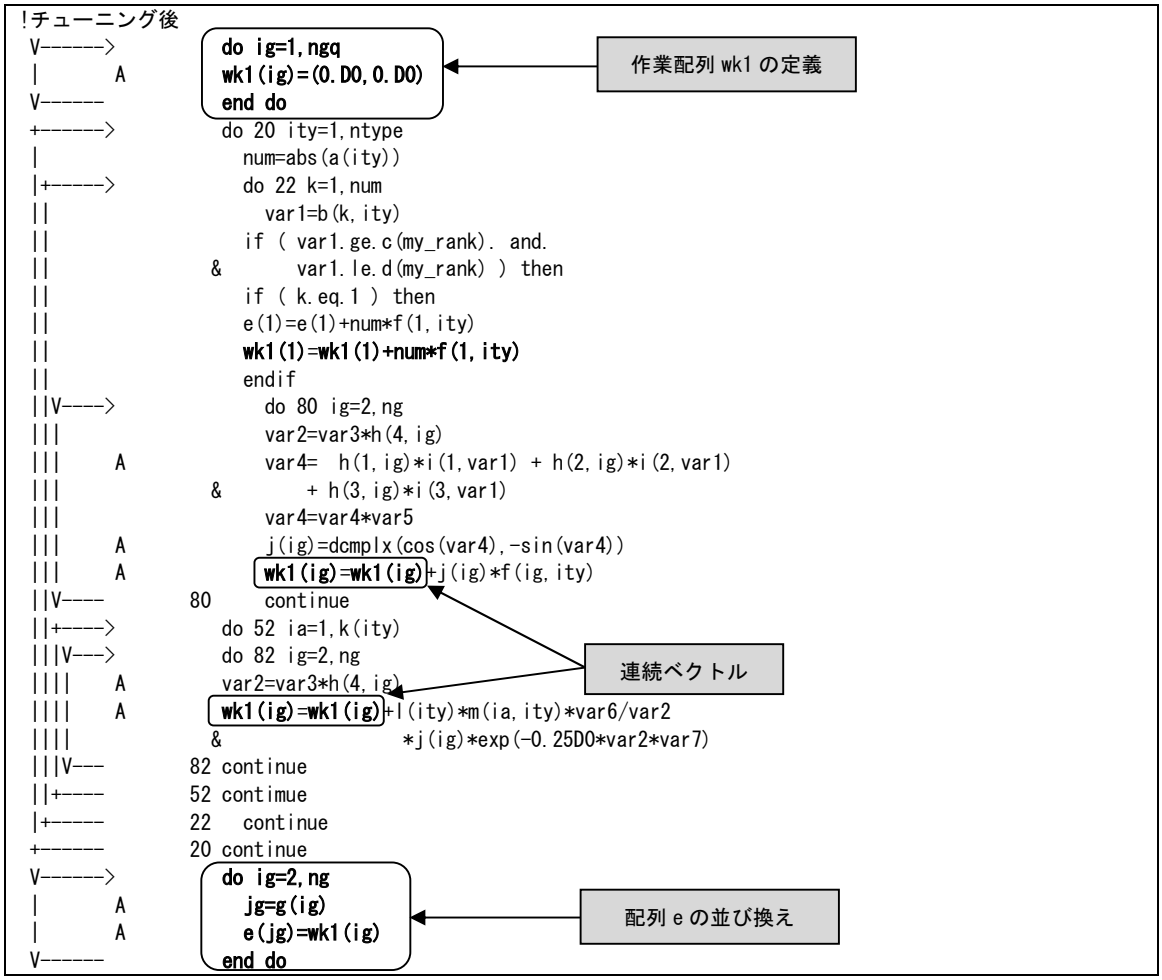


図 6.1.13-3 リストアクセス削減後のコード

(3) 性能分析

図 6.1.13-4 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
96096	12984.005(9.5)	135.115	10777.3	5016.0	99.40	253.6	11083.951	13.631	4.866	**215.473**	**3830.030**	5.48	【チューニング前】
96096	5698.233(4.5)	59.297	22552.9	11429.4	99.38	253.8	4290.098	2.336	5.066	**49.927**	**1148.920**	2.14	【チューニング後】

図 6.1.13-4 リストアクセス削減前後の FTRACE 情報

リストアクセスを削減することでメモリ負荷が減少し, CPU ポート競合時間が 215.5 秒から 49.9 秒に, メモリネットワーク競合時間が 3830.0 秒から 1148.9 秒にそれぞれ削減され, 実行時間が 12984.0 秒から 5698.2 秒に短縮されている.

6.1.14. 外側ループのアンロールによるメモリアクセス効率化

(1) チューニング方針

多重 DO ループでステンシル計算になっている場合，多重ループのアウターアンロールによってメモリアクセスの削減を図れることが多い．本事例ではコンパイル指示行 OUTERUNROLL による最適化事例を示す．

図 6.1.14-1 にチューニング前のコードを示す．二重 DO ループにおいて， i のインデックスを持つ内側の DO ループはベクトル化されている．図 6.1.14-2 に示したチューニング前の FTRACE 情報では，ベクトル演算率，平均ベクトル長ともに十分な値ではあるが，メモリネットワーク競合時間の値が実行時間に対して約 14% と割合が大きい．外側 DO ループのアンローリング（アウターアンロール）により，メモリアクセス回数を削減してバンクコンフリクトの時間を短縮することを検討する．

内側の DO ループ内では配列 b の 2 次元目において， j-1, j および j+1 が参照される．配列 b は a, c, d, e の演算時に参照されるので，インデックス j を持つ外側の DO ループをアンロール（アウターアンロール）することで，メモリアクセス回数が削減される．SX-ACE では最適化オプションが「vopt」以上の場合，コンパイラが自動で判断し外側ループのアンロールを行うが，アウターアンロールによるオーバーヘッドを考慮した結果，自動ではアウターアンロールが実施されない場合がある．ただし，コンパイル時には不明な情報もあるため，コンパイラがアウターアンロールを実施しない場合であっても，実際にはアウターアンロールにより高速化されることもある．その場合，本コードのように手動で指示行を挿入する．外側ループがアンロールされたことを確認するには，コンパイラオプション「-R2」を指定することで変形リストを出力する．アウターアンロールする段数 N の値は 2 のべき乗を指定するが，N の値が大きすぎると性能低下の恐れがあるので注意が必要である．

```
!チューニング前
 +-----> do   j=1,jm-1
 |V----->    do  i=-im,im
 ||
 ||     A    a(i,j)=(b(i+1,j)+b(i-1,j))/(var1**2)
 ||     A    c(i,j)=(b(i,j+1)-b(i,j-1))/(2*var1)
 ||     A    d(i,j)=(b(i,j+1)+b(i,j-1))/(var1**2)
 ||     A    e(i,j)=(b(i+1,j+1)-b(i+1,j-1)-b(i-1,j+1)+b(i-1,j-1)) /(4.0D0*var1**2)
 (略)
 |V----      end do
 +-----    end do
```

図 6.1.14-1 外側ループのアンロール前のコード

```
FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS V.OP  AVER.    VECTOR I-CACHE O-CACHE   BANK CONFLICT    ADB HIT  PROC.NAME
           TIME[sec](  %  )   [msec]                    RATIO V.LEN     TIME    MISS    MISS CPU PORT  NETWORK   ELEM.%

    18000     4.493(  5.5)     0.250  49232.3  23601.9 99.74 200.8     4.479   0.001   0.002    0.241    0.635    62.66 【チューニング前】
```

図 6.1.14-2 外側ループのアンロール前の FTRACE 情報

(2) チューニング内容

図 6.1.14-3 にチューニング後のコードと変形リスト示す．2 次元目の DO ループの前に外側ループの 4 段アンローリング行う指示行（!CDIR OUTERUNROLL=4）を挿入することで，コンパイラが

アウターアンロールを行う. アウターアンロールにより配列 b についてメモリアクセス回数を1/3に削減することができる.

```
!チューニング後
       !cdir outerunroll=4          <-- 指示行の挿入
 +-----> do  j=1,jm-1
 |V----->    do  i=-im,im
 ||
 ||     A    a(i,j)=(b(i+1,j)+b(i-1,j))/(var1**2)
 ||     A    c(i,j)=(b(i,j+1)-b(i,j-1))/(2*var1)
 ||     A    d(i,j)=(b(i,j+1)+b(i,j-1))/(var1**2)
 ||     A    e(i,j)=(b(i+1,j+1)-b(i+1,j-1)-b(i-1,j+1)+b(i-1,j-1)) /(4.0D0*var1**2)
 (略)
 |V----     end do
 +-----   end do

! 変形リストの抜粋
   (j=1,2 のときの処理 : 省略)
   (j=3～74 までの処理を 4 飛びでアンローリング)
    .  do j = 3, 74, 4
    .     d14 = 1.D0/1.77777777777777e-004
    .     d15 = 1.D0/2.66666666666666e-002
    .     d16 = 1.D0/1.77777777777777e-004
    .     d17 = 1.D0/7.11111111111111e-004
 (略)
    .    do i = 1, 601
    .      a(i-301,j  ) = (b(i-300,j  )+b(i-302,j  ))*d14
    .      a(i-301,j+1) = (b(i-300,j+1)+b(i-302,j+1))*d14
    .      a(i-301,j+2) = (b(i-300,j+2)+b(i-302,j+2))*d14
    .      a(i-301,j+3) = (b(i-300,j+3)+b(i-302,j+3))*d14
    .      c(i-301,j  ) = (b(i-301,j+1)-b(i-301,j-1))*d15
    .      c(i-301,j+1) = (b(i-301,j+2)-b(i-301,j  ))*d15
    .      c(i-301,j+2) = (b(i-301,j+3)-b(i-301,j+1))*d15
    .      c(i-301,j+3) = (b(i-301,j+4)-b(i-301,j+2))*d15
    .      d(i-301,j  ) = (b(i-301,j+1)+b(i-301,j-1))*d16
    .      d(i-301,j+1) = (b(i-301,j+2)+b(i-301,j  ))*d16
    .      d(i-301,j+2) = (b(i-301,j+3)+b(i-301,j+1))*d16
    .      d(i-301,j+3) = (b(i-301,j+4)+b(i-301,j+2))*d16
    .      d(i-301,j  ) = (b(i-300,j+1)-b(i-300,j-1)-b(i-302,j+1)+b(i-302,j-1))*d17
    .      d(i-301,j+1) = (b(i-300,j+2)-b(i-300,j  )-b(i-302,j+2)+b(i-302,j  ))*d17
    .      d(i-301,j+2) = (b(i-300,j+3)-b(i-300,j+1)-b(i-302,j+3)+b(i-302,j+1))*d17
    .      d(i-301,j+3) = (b(i-300,j+4)-b(i-300,j+2)-b(i-302,j+4)+b(i-302,j+2))*d17
 (略)
    .    enddo
    .  enddo
```

図 6.1.14-3 外側ループのアンロール後のコードと編集リスト

(3) 性能分析

図 6.1.14-4 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
18000	4.493(5.5)	0.250	49232.3	23601.9	99.74	200.8	4.479	0.001	0.002	0.241	0.635	62.66	【チューニング前】
18000	3.760(11.1)	0.209	57297.0	26787.0	99.76	201.8	3.734	0.008	0.008	0.258	0.356	54.73	【チューニング後】

図 6.1.14-4 外側ループのアンロール前後の FTRACE 情報

ベクトル長および平均ベクトル長にはほぼ変化はないが, 実行時間に対するメモリネットワーク競合時間の割合は約 9.5% に減少し, 実行時間が 4.5 秒から 3.8 秒に短縮されている.

6.1.15. 配列のサイズ変更によるバンクコンフリクト時間の削減

(1) チューニング方針

5.2.1 節で示したようにメモリからデータを読み込むときにバンクコンフリクトが発生する場合がある. 本事例では, 配列のサイズを変更してバンクコンフリクトを回避する例を示す.

図 6.1.15-1 にチューニング前のコードを示す. 本コードでは, FFT ライブラリとして SX-ACE に最適化された ASL ライブラリが用いられている. ASL ライブラリでは配列の 1 次元目のサイズが 2 の累乗の場合, バンクコンフリクトの増加により性能が著しく低下することがある. 配列の 1 次元目のサイズが 2 の累乗となる場合, 配列の 1 次元目のサイズを +1 に変更するか, 1 次元目の整合寸法を「2 の累乗+1」とした作業配列を定義し, 本来の配列から要素の値をコピーすることになる. ただし, 作業配列を使用する場合, ASL ライブラリを呼び出した後も, 作業配列から元の配列に結果をコピーする必要があるためメモリコピーがオーバーヘッドとなる.

図 6.1.15-2 に配列サイズの変更前の FTRACE 情報を示す. 本コードでは ASL ライブラリの引数となる配列 a の 1 次元目のサイズが $64 = 2^6$ であるので, メモリネットワーク競合時間の値が実行時間の約 82%を占めている. 本コードでは他の演算を行う箇所では, DO ループの始値と終値が明示的に指定されているので, 1 次元目の配列のサイズを +1 に変更して定義することで, ASL ライブラリ実行時のバンクコンフリクト時間を短縮することが可能である.

```
!チューニング前
! 配列宣言部分
      integer,parameter ::n=64,wm=2048
      complex(8)::a(0:n-1,0:n-1,0:wm-1)
! ASL 呼び出し部分
      call ZFC3FB(n, n, wm, a, n, n, wm, isw, ifax, trigs, wk, ierr)
```

図 6.1.15-1 配列サイズ変更前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
142	138.112(22.4)	972.621	1408.9	659.4	98.56	213.6	137.976	0.001	0.051	2.425	**112.758**	33.33	【チューニング前】

図 6.1.15-2 配列サイズ変更前の FTRACE 情報

(2) チューニング内容

図 6.1.15-3 にチューニング後のコードを示す. 配列の宣言部分で, 1 次元目の配列サイズが n = 64 であった箇所を n = 65 に変更した. ASL ライブラリの引数の第 1～3 引数は 1～3 次元目のデータ数, 第 4 引数は FFT を行う入出力配列, 第 5 引数は 1 次元目の整合寸法, 第 6, 7 引数は 2, 3 次元目の寸法であるので, 実際のデータが保存された範囲として 1, 2 次元目のサイズが n-1 = 64 となるように第 1, 2 引数を変更した.

```
!チューニング後
! 配列宣言部分
   integer,parameter :: n=65, wm=2048
   complex(8)::a(0:n-1,0:n-1,0:wm-1)
! ASL 呼び出し部分
      call ZFC3FB(n-1, n-1, wm, a, n, n, wm, isw, ifax, trigs, wk, ierr)
```

n の値を 65 に変更

ASL ライブラリの引数を変更

図 6.1.15-3 配列サイズ変更後のコード

(3) 性能分析

図 6.1.15-4 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
142	138.112(22.4)	972.621	1408.9	659.4	98.56	213.6	137.976	0.001	0.051	2.425	**112.758**	33.33	【チューニング前】
142	7.998(8.1)	56.725	43620.8	17917.9	98.37	230.1	7.895	0.001	0.038	0.177	**2.123**	70.10	【チューニング後】

図 6.1.15-4 配列サイズ変更前後の FTRACE 情報

1 次元目の配列サイズの変更により, ASL ライブラリ呼び出し部分のベクトル演算率, 平均ベクトル長ともに大きな変化はないが, メモリネットワーク競合時間の値が大きく減少し, 実行時間が 138.1 秒から 8.0 秒に短縮されている.

6.1.16. 作業配列の導入による I/O 時間の短縮

(1) チューニング方針

効率の良いファイルアクセス方法は, 連続的に並んでいるデータを複数要素の塊として入出力することである. 処理によっては不連続なデータの読み込みが必要で, 1 要素ごとにデータを読み込む場合もある. 本事例はそのような場合における高速化手法を示している.

図 6.1.16-1 にチューニング前のコードを示す. READ 文によって配列 wk1 にデータを読み込んでいる. このときインデックス配列 a を用いて 1 次元目の位置を決定している. この場合, ファイル読み込みが 1 要素ずつとなるため処理時間が増加する. 図 6.1.16-2 はチューニング前の FTRACE 情報であるが, ベクトル演算率は 2.5% と低く, またインデックス配列を参照するためメモリネットワーク競合時間の値が大きくなっている.

本事例では作業配列を導入し, バイナリ形式のファイルから配列要素を連続的に読み込み, その後に, インデックス配列を参照して本来の配列に値をストアすることを検討した.

```
!チューニング前                                                    ファイルの読み込み時にインデックス配列を使用
 ||||              if(mype.eq.0)then
 |||+--->             do ib=1,ib_end
 ||||    A                 read(iunit,iostat=ios) (wk1(a(ig),ib) ig=1,ngl)
 ||||                      if(ios.gt.0)then
 ||||                        write(*,'(a,i4)')'there can be read error!? ios=',ios
 ||||                        call mpi_finalize(ierr)
 ||||                        stop
 ||||                      end if
 |||+---               enddo
 ||||              endif
```

図 6.1.16-1 作業配列導入前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
1	1761.595(60.8)	1761595.187	1028.4	0.0	**2.50**	4.5	472.525	0.655	7.702	0.023	74.193	1.49	【チューニング前】

図 6.1.16-2 作業配列導入前の FTRACE 情報

(2) チューニング内容

図 6.1.16-3 にチューニング後のコードを示す. チューニング前の READ 文では配列 wk1 の 1 次元目の要素をインデックス配列 a により指定しているが, 本修正においては READ 文で作業配列 wk2 に連続的にデータを読み込み, 次の DO ループでインデックス配列 a を用いて wk2 から wk1 に値をストアするようにした. これにより I/O が効率的に行われるとともに, 値をストアする DO ループではベクトル化が可能となっている.

```
!チューニング後
! 作業配列の定義
          complex(8), allocatable :: wk2(:,:)
! 作業配列の確保
          allocate(wk2(ngl,ib_end))
  :
(省略)
  :
 |||                    if(mype.eq.0)then
 |||+--->                 do ib=1,ib_end
 ||||                       read(iunit,iostat=ios) (wk2(ig,ib),ig=1,ngl)
 ||||                       if(ios.gt.0)then
 ||||                         write(*,'(a,i4)')'there can be read error!? ios=',ios
 ||||                         call mpi_finalize(ierr)
 ||||                         stop
 ||||                       end if
 |||+---                  enddo
! インデックス配列を参照して, ストアする
 |||+--->                 do ib=1,ib_end
 ||||V-->                   do ig=1,ngl
 |||||   A                    wk1(a(ig),ib)=wk2(ig,ib)
 ||||V--                    end do
 |||+---                  end do
 |||                    endif
```

作業配列に連続で読み込む

インデックス配列を参照してストア

図 6.1.16-3 作業配列導入後のコード

(3) 性能分析

図 6.1.16-4 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
1	1761.595(60.8)	1761595.187	1028.4	0.0	**2.50**	4.5	472.525	0.655	7.702	0.023	74.193	1.49	【チューニング前】
1	23.053(1.9)	23053.243	2882.1	0.2	**80.16**	185.0	10.125	0.511	2.682	0.065	2.079	12.35	【チューニング後】

図 6.1.16-4 作業配列導入前後の FTRACE 情報

インデックス配列を参照して値をストアする部分の 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 2.5% から 80.2% に向上し, I/O を含めた実行時間は 1,761.6 秒から 23.1 秒に短縮されている.

6.1.17. ファイル形式の変更による I/O の高速化

(1) チューニング方針

SX-ACE ではファイルの入出力を行う場合, バイナリ形式のファイルを取り扱う方が, 書式付きテキスト形式のファイルよりも, 格段に高速な入出力が可能である. 本事例では, ファイルのバイナリ形式化と演算とファイルの書き出しを分離し, 演算処理のベクトル化を行う例を示す. チュー

ニング前のコードを図 6.1.17-1 に示す. 本コードでは複数の演算結果の配列データを個々に, 書式付きテキスト形式で一定ステップ毎に書き出している. また, DO ループ内で演算を行うとともに, WRITE 文によるファイルへの書き出しも行っている. 図 6.1.17-2 にチューニング前の FTRACE 情報を示す. DO ループ内に WRITE 文があるために, 二重 DO ループはベクトル化対象外となっており, ベクトル演算率はほぼ 0% である.

作業配列を導入して演算部分を先に実行することでベクトル化を促進し, また書き出される複数のファイルを, 書式付きテキスト形式から, バイナリ形式の 1 ファイルにまとめて書き出すことにより出力の高速化を行う.

```
!チューニング前
 +------>       do  i=-im-1,im+1
 |+----->       do  j=0,jm
 ||               var1=sqrt(a(i,j)**2+b(i,j)**2)
 ||               write(110,1001) c(i), d(i,j), e(i,j)
 ||               write(112,1002) c(i), d(i,j), f(i,j)
 ||               write(113,1003) c(i), d(i,j),var1,a(i,j),b(i,j)
 |+-----        end do
 |                write(110,*) c(i), '', 3.0,  -''
 |                write(112,*) c(i), '', 3.0,  -''
 |                write(113,*) c(i), '', 3.0,  -,  0,  0''
 +------         end do
```

図 6.1.17-1 ファイル形式変更前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
20	22.710(2.1)	1135.502	755.9	11.4	**0.28**	122.8	0.012	2.006	2.025	0.000	0.001	73.95	【チューニング前】

図 6.1.17-2 ファイル形式変更前の FTRACE 情報

(2) チューニング内容

図 6.1.17-3 にチューニング後のコードを示す. 演算結果を格納する作業配列 wk1 を定義し, 事前に演算を行う. 書き出す 9 つの配列は 1 つのバイナリ形式のファイルに同時に書き出すことで, 出力の高速化が可能となる. ただし, ポスト処理で結果ファイルを読み込む場合も同型式で読み込む必要がある.

```
!チューニング後
  ! 作業配列の定義
              real*8::wk1(ln,aa)
  ! バイナリファイルを書き出し用に OPEN
|             open(200,file="RESULT_"//trim(STEP)//".bin",form='unformatted')
  ! 演算部分
 +------>     do  j=0,jm
 |V----->       do  i=-im-1,im+1
 ||               wk1(i,j)=sqrt(a(i,j)**2+b(i,j)**2)
 |V-----        end do
 +------       end do
  ! バイナリ形式で書き出し
              write(200) ln,aa,c,d,e,f,wk1,a,b
```

演算結果を作業配列に格納

図 6.1.17-3 ファイル形式方法変更後のコード

(3) 性能分析

図 6.1.17-4 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
20	22.710(2.1)	1135.502	755.9	11.4	**0.28**	122.8	0.012	2.006	2.025	0.000	0.001	73.95	【チューニング前】
20	0.029(0.1)	1.439	2281.0	169.6	**70.51**	231.5	0.002	0.003	0.004	0.000	0.000	1.71	【チューニング後】

図 6.1.17-4 ファイル形式方法変更前後の FTRACE 情報

書き出しに必要な変数の演算部分を, 作業配列を導入することでベクトル化可能となり, ベクトル演算率は 70.5% まで増加した. また書き出しファイルのバイナリ形式化と一本化により, 実行時間は 22.7 秒から 0.029 秒に短縮することができ, 計算結果を書き出すための実行時間を大幅に短縮することができた. 結果の検証や可視化についても, サイズの大きいテキスト形式のファイルを扱うよりも, バイナリ形式のファイルを取り扱うことで, 入出力時間を短縮することも可能となっている.

6.1.18. 複合問題の事例

(1) チューニング方針

一般に, 一つの DO ループに対して複数のチューニングを行うことはよくあることである. 本事例では, 一つの DO ループに対して 3 種類のチューニングを段階的に行う例を示す.

図 6.1.18-1 にチューニング前のコードを示す. 最内ループでベクトル化が行われているが, ベクトル化された DO ループのループ長が 9, 10 と非常に短くなっている. また, その外側のインデックス m の DO ループに着目すると, ループからの飛び出しが存在する. ループからの飛び出しがある場合, ループ回数が不定となるためベクトル化を阻害する要因となる. 図 6.1.18-2 の FTRACE 情報では, ベクトル演算率が 1.10 % と非常に低く, 平均ベクトル長が 8.4 と十分なベクトル長を確保できていない. このような場合のチューニング手順について段階に分けて後述する.

```
!チューニング前
 +-----> do k = kks, kkm
 |V----> do j = kjs, kjm
 ||+---->   do m =1, 10000000
 |||V-->      do i=5,13
 ||||             演算部 A
 ||||
 |||V--       end do
 |||
 |||V-->      do i=5,13
 ||||             演算部 B
 ||||
 |||V--       end do
 |||
 |||+-->      do i=1,10
 ||||             演算部 C
 ||||
 |||+--       end do
 |||          if (var1 >= var2) exit   <-- ループからの飛び出し
 ||+--      end do
 ||V--->    do i=5,13
 |||            演算部 D
 |||
 ||V--->    end do
```

```
 |V----->  end do
+------>  end do
```

図 6.1.18-1 チューニング前のコード

```
FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS V.OP  AVER.    VECTOR I-CACHE O-CACHE   BANK CONFLICT    ADB HIT  PROC.NAME
           TIME[sec](  %)    [msec]                   RATIO V.LEN      TIME    MISS    MISS CPU PORT  NETWORK   ELEM.%

     1000 16098.249( 33.1) 16098.249    839.3    432.8  1.10   8.4    538.965  0.650 519.209    0.036    0.051    95.93 【チューニング前】
```

図 6.1.18-2 チューニング前の FTRACE 情報

(2) チューニング内容

＜ステップ 1 EXPAND 指示行によるループ展開＞

最内ループのループ長が非常に短いため，EXPAND 指示行によりループ展開を行い，外側の DO ループでベクトル化を行うことでループ長を拡大する．チューニング後のコードを図 6.1.18-3 に示す．EXPAND 指示行を挿入することで，最内ループが展開されている(V マークがなくなり＊マークとなっている)ことが確認できる．

＜ステップ 2 ループブロッキング＞

最内ループをループ展開することで，インデックス m の DO ループが最内側となるが，前述のとおり， m の DO ループにはループからの飛び出しが存在するため，このままの形ではベクトル化することができない．そこで j のループをブロッキングし，ブロック化された DO ループと m の DO ループを入れ替え，ブロック化された DO ループでベクトル化を行う．チューニング後のコードを図 6.1.18-3 に示す．

ブロッキングサイズとして， blksz にはベクトルレジスタ長と同じ 256 を指定する．ループ長がベクトルレジスタ長を超えないことが明白であるため，ブロック化された DO ループには SHORTLOOP 指示行を指定する．

＜ステップ 3 マスク処理によるループからの飛び出しへの対応＞

チューニング前のコードでは，インデックス j の DO ループの内側に m の DO ループがあり，m の DO ループの計算において収束判定を行っている．収束判定が真になると m の DO ループから飛び出し， j の値を更新した後に m の DO ループの計算を再度実行している．すなわち，j の値それぞれに対して， m の DO ループで収束判定が行われている．

チューニング後のコードを図 6.1.18-3 に記載する．チューニング後のコードでは， m の DO ループの内側にブロック化された j の DO ループが存在し， j の DO ループでベクトル化されている．そこで，ブロック化された DO ループ内において， j の値ごとに収束状況を保持する作業配列(wk1)を導入し，収束したと判断された時点で，この作業配列に収束済みフラグを設定する(wk1 配列に 1 を設定する)．j の DO ループの最初で収束済みフラグの確認を行っており，収束済みフラグが設定されたデータについてはマスク処理により計算を行わないようにしている．ブロック化された DO ループ内のすべての j について収束すると， m の DO ループを抜け，次のブロックの処理に移る．

```
!チューニング後
  +-----> do k = kks, kkm
  |+----->  do jj = kjs, kjm, blksz  <-- ブロッキング
  ||       !cdir shortloop
  ||V--->   do j = jj, min(jj+blksz-1, kjm)
  ||V---    end do
  ||        !cdir shortloop
  ||V--->   do j= 1, blksz
  |||        wk1(j)=0               <-- 作業配列
  ||V---    end do
  ||+--->   do m=1,10000000
  |||        !cdir shortloop
  |||V-->    do j = jj, min(jj+blksz-1, kjm)
  ||||         l=j-jj+1
  ||||         if(wk1(l) == 0) then  <-- マスク処理
  ||||         !cdir expand=9
  ||||*->      do i=5,13
  ||||          演算部 A
  ||||
  ||||*-       end do
  ||||         !cdir expand=9
  ||||*->      do i=5,13
  ||||          演算部 B
  ||||
  ||||*-       end do
  ||||         !cdir expand=10
  ||||*->      do i=1,10
  ||||          演算部 C
  ||||
  ||||*--      end do
  ||||         if (a(l) >= b(l) ) wk1(l)=1   <-- 収束判定
  ||||         end if
  |||V--     end do
  |||
  |||        finished = 1
  |||        !cdir shortloop
  |||V-->    do j = jj, min(jj+blksz-1,kjm)
  ||||        l=j-jj+1
  ||||        if(wk1(l) == 0) finished = 0
  |||V--     end do
  |||        if(finished == 1) exit  <-- ループからの飛び出し
  |||
  ||+---    end do
  ||         !cdir shortloop
  ||V--->    do j = jj, min(jj+blksz-1, kjm)
  |||        !cdir expand=9
  |||*--->   do i=5,13
  |||         演算部 D
  |||
  |||*---     end do
  ||V---     end do
  |+----    end do
  +-----   end do
```

図 6.1.18-3 チューニング後のコード

(3) 性能分析

図 6.1.18-4 にチューニング前後の FTRACE 情報を示す.

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CPU PORT	CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
1000	16098.249(33.1)	16098.249	839.3	432.8	**1.10**	**8.4**	538.965	0.650	519.209	0.036	0.051	95.93	【チューニング前】
1000	825.157(73.3)	825.157	44075.2	16195.1	**99.86**	**239.0**	803.505	0.026	0.003	0.528	92.972	57.5	【チューニング後】

図 6.1.18-4 チューニング後の FTRACE 情報

本チューニングによりベクトル演算率が 1.1 %から 99.9 %に，平均ベクトル長が 8.4 から 239.0 まで向上した．最適化によりベクトルプロセッサの性能を最大限に引き出すことができ，実行時間が 16,098.2 秒から 825.2 秒に短縮されている．

6.2. LX406Re-2 のノード内最適化事例

6.2.1. インライン展開による高速化

(1) チューニング方針

一般にサブルーチンの呼び出し回数が多いと，そのサブルーチンへ処理を移すためのオーバーヘッドが顕著になり，処理時間が長くなることがある．本事例は，呼び出し回数の多いサブルーチンをインライン展開し，そのオーバーヘッド時間を削減する例である．

インテルコンパイラでは，コンパイラオプション「-p」を指定することで，サブルーチン単位のプロファイル情報を採取することができる．図 6.2.1-1 にチューニング前のプロファイル情報を示す．計算コスト上位に呼び出し回数の多いサブルーチンが多数ある．サブルーチン呼び出しのオーバーヘッドが累積し，コスト増加につながっていると考えられる．インライン展開によりサブルーチン呼び出しのオーバーヘッドを削減し，高速化を図る．

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
24.88	7.99	7.99	1000	7.99	21.21	sub1
18.84	14.04	6.05	145797084	0.00	0.00	sub2
7.50	16.45	2.41	111388943	0.00	0.00	sub3
5.95	18.36	1.91	51533246	0.00	0.00	sub4
3.27	19.41	1.05	142121592	0.00	0.00	sub5
2.86	20.33	0.92	1000	0.92	2.16	sub6
2.68	21.19	0.86	23799014	0.00	0.00	sub7
2.68	22.05	0.86	3000	0.29	0.29	sub8
2.65	22.90	0.85	3002	0.28	0.28	sub9
2.52	23.71	0.81	4000	0.20	0.20	sub10
2.46	24.50	0.79	16955255	0.00	0.00	sub11
2.15	25.19	0.69				sub12
1.87	25.79	0.60				sub13
1.25	26.19	0.40	1000	0.40	0.40	sub14
1.23	26.59	0.40	18620162	0.00	0.00	sub15
1.06	26.93	0.34	1664907	0.00	0.00	sub16
1.06	27.27	0.34	1000	0.34	1.62	sub17
0.97	27.58	0.31				sub18
0.97	27.89	0.31				sub19
0.93	28.19	0.30	1000	0.30	26.30	sub20
0.83	28.45	0.27	16938376	0.00	0.00	sub21
0.81	28.71	0.26	4000	0.07	0.07	sub22
0.78	28.96	0.25	480288	0.00	0.00	sub23
0.72	29.19	0.23				sub24
0.65	29.40	0.21				sub25
0.62	29.60	0.20				sub26
0.58	29.79	0.19	48441787	0.00	0.00	sub27
0.56	29.97	0.18	9000000	0.00	0.00	sub28
0.50	30.13	0.16	7155965	0.00	0.00	sub29
0.50	30.29	0.16	33910510	0.00	0.00	sub30

図 6.2.1-1 インライン展開前のプロファイル情報

(2) チューニング内容

コンパイラオプションに「-ipo」を指定することにより，コンパイラが自動的にインライン展開を

行う. 対象は, 同一ファイル内および別のファイル内で定義されているサブルーチンへの呼び出しである. 同一ファイル内の一部のサブルーチンはコンパイラオプションで自動展開されないため, forceinline 指示行の挿入によりインライン展開を行う. しかし, 多数のサブルーチンをインライン展開してしまうと, 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あふれてしまい, プログラム全体の実行性能が低下することがある. そのため, 性能向上の効果がみられるサブルーチンのみを展開する. 指示行を追加したソースコードの例を図 6.2.1-2 に示す.

```
// コンパイラ指示行の挿入
#pragma forceinline recursive
sub2( var1, a[i], var2, var3, var4, var5, var6, var7, var8, var9 );
```

図 6.2.1-2 指示行追加後のコード

(3) 性能分析

図 6.2.1-3 にチューニング後のプロファイル情報を示す. ここでは, sub2, sub7, sub21 をインライン展開している.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
39.55	10.18	10.18	1000	10.18	17.81	sub1
12.59	13.42	3.24	18620162	0.00	0.00	sub15
6.72	15.15	1.73	1000	1.73	1.98	sub6
5.48	16.56	1.41	31267185	0.00	0.00	sub4
4.16	17.63	1.07	142121592	0.00	0.00	sub5
3.85	18.62	0.99	1000	0.99	2.42	sub31
3.50	19.52	0.90	3000	0.30	0.30	sub8
3.26	20.36	0.84	16955255	0.00	0.00	sub11
2.72	21.06	0.70	1000	0.70	1.06	sub17
1.94	21.56	0.50	1000	0.50	22.16	sub20
1.75	22.01	0.45	1000	0.45	0.45	sub14
1.57	22.42	0.41	2178852	0.00	0.00	sub32
1.32	22.76	0.34	1664907	0.00	0.00	sub16
1.20	23.07	0.31				sub24
1.01	23.33	0.26	1000	0.26	0.26	sub9
0.89	23.56	0.23				sub25
0.72	23.74	0.19	33910510	0.00	0.00	sub30
0.70	23.92	0.18	512320	0.00	0.00	sub23
0.66	24.09	0.17	4289808	0.00	0.00	sub33
0.62	24.25	0.16	3762443	0.00	0.00	sub34
0.58	24.40	0.15	3003000	0.00	0.00	sub35
0.58	24.55	0.15				sub36
0.51	24.68	0.13				sub37
0.43	24.79	0.11	1706706	0.00	0.00	sub38
0.39	24.89	0.10	16955255	0.00	0.00	sub39
0.35	24.98	0.09	1	90.00	90.10	sub40
0.35	25.07	0.09	1087712	0.00	0.00	sub41
0.31	25.15	0.08	1089426	0.00	0.00	sub42
0.31	25.23	0.08	1000	0.08	0.08	sub43
0.23	25.29	0.06	2736870	0.00	0.00	sub44

図 6.2.1-3 インライン展開後のプロファイル情報

インライン展開されたサブルーチンが, 呼び出し元のサブルーチンに吸収されたことがわかる. サブルーチン呼び出しのオーバーヘッドが削減され, 実行時間が 54 秒から 40 秒に短縮された.

6.3. MPI 最適化事例

6.3.1. メモリ使用量の削減

(1) チューニング方針

MPI のプログラムにおいて，プログラム実行の途中または終了時に，各プロセスが保持しているデータを 1 つのプロセスに集めて結果を出力する場合がある．本事例では，モデル全体のデータを格納する配列をデータ出力するプロセスのみが確保するようにしてメモリ使用量の削減を行う例を示す．MPI プログラムによる大規模な計算を行う場合，各計算ノードで使用可能なメモリ使用量は限られているため，メモリ使用量を削減することは重要である．

図 6.3.1-1 にチューニング前のコードを記載する．サブルーチン gather_root は，計算元となるプロセスからデータをランク 0 番のプロセスに集める MPI_GATHER 通信の役割を果たしている．モデル全体のデータを集める配列は静的に確保されており，全てのプロセスでモデル全体のデータ量を確保するようになっている．図 6.3.1-2 にチューニング前の PROGINF 情報のメモリ使用量を記載する．

```
!チューニング前
      use mpi
      use mpisub
      include "subcom_g.inc"
      character(len=*) :: fname
      real*8,dimension(if,jp,kf,mf) :: a
      real*8,dimension(if,jf,kf) :: b, c, d, e, f
  :
(省略)
  :
      call gather_root(g,a,0,displs,counts,inum,if,jf,kf,mf)
```

図 6.3.1-1 メモリ使用量の削減前のコード

```
      【チューニング前】
      Memory Size (MB)          :        19072.031250   (ランク 0 番)
      Memory Size (MB)          :        19072.031250   (ランク 1 番)
```

図 6.3.1-2 メモリ使用量削減前の PROGINF 情報(メモリ使用量)

(2) チューニング内容

図 6.3.1-3 にチューニング後のコードを記載する．モデル全体のデータを集める配列は ALLOCATABLE 属性として，ランク 0 番のみ必要な大きさを確保するように修正する．使用しないランクではダミーで大きさ 1 の配列を確保している．チューニングにより，不要なメモリ使用量の削減が可能となる．

```
!チューニング後
      use mpi
      use mpisub
      include "subcom_g.inc"
      character(len=*) :: FNAME
      real*8,allocatable,dimension(:,:,:,:) :: a
      real*8,allocatable,dimension(:,:,:) :: b, c, d
      real*8,allocatable,dimension(:,:,:) :: e, f
  :
(省略)
  :
```

```
      if (needoutput) THEN
        allocate(a(if,jf,kf,mf))
        allocate(b(if,jf,kf),c(if,jf,kf),d(if,jf,kf))
        allocate(e(if,jf,kf),f(if,jf,kf))
      else
        allocate(a(1,1,1,mf))
        allocate(b(1,1,1),c(1,1,1),d(1,1,1))
        allocate(e(1,1,1),f(1,1,1))
      endif
 :
(省略)
 :
      call gather_root(g,a,0,displs,counts,inum,if,jf,kf,mf)
```

配列を使用するランクでは必要な大きさを確保

配列を使用しないランクではダミー配列とする

図 6.3.1-3 メモリ使用量の削減後のコード

(3) 性能分析

図 6.3.1-4 にチューニング前後の性能値を示す.

```
【チューニング前】
Memory Size (MB)          :       19072.031250  (ランク 0 番)
Memory Size (MB)          :       19072.031250  (ランク 1 番)
【チューニング後】
Memory Size (MB)          :       11392.031250  (ランク 0 番)
Memory Size (MB)          :        3136.031250  (ランク 1 番)
```

図 6.3.1-4 メモリ使用量削減前後の PROGINF 情報(メモリ使用量)

チューニング後はモデル全体のデータを集めるランク 0 番と他のランクでメモリ使用量に差が出ている. ランク 1 番では, メモリ使用量を 19.1GB から 3.1GB に削減することができている. また, チューニング前のコードのように配列を静的に確保している場合, 入力データの違いなどで実際には実行されないサブルーチンがあったとしても, コンパイル時にコンパイル対象のすべてのサブルーチンについて必要なメモリサイズの確保を行う. 実際には実行されないサブルーチンが同様の構造をしている場合, 静的配列から動的配列に変更することで, 該当のサブルーチンが実行されない場合にはメモリの確保は行われないため, 集める先となるランク 0 番でもメモリ使用量を削減することができる.

6.3.2. リダクション処理の変更による高速化

(1) チューニング方針

MPI のプログラムにおいて, プログラム全体の総和や最大値を求める場合がある. 本事例では, 計算に必要な全てのデータをランク 0 番に収集し, ランク 0 番で総和や最大値を求めるのではなく, 各プロセスで求めた部分和・部分最大値を通信することで, 計算コストやメモリ使用量の分散化を行う方法を示す.

図 6.3.2-1 にチューニング前のコードを記載する. ここでのサブルーチン gather_root は, 計算元となるプロセスからデータをランク 0 番のプロセスに集める MPI_GATHER 通信の役割を果たしている. ランク 0 番のプロセスは, 集まったモデル全体のデータを用いて総和・最大値を求めている. 図 6.3.2-2 にチューニング前の FTRACE 情報を, 図 6.3.2-3 にチューニング前の PROGINF 情報のメモリ使用量を記載する.

```
!チューニング前
      subroutine sub1(a, b)
      use mpisub
      include "subcom_g.inc"
      real*8,dimension(mf) :: a, b
      real*8,dimension(inum,jf,jk) :: c
      real*8,dimension(if,jk,jk) :: d
      real*8 :: var1
C
      do m=1,mf
        a(m) = 0.0D0
        b(m) = 0.0D0
  :
（省略）
  :
        call gather_root (c, d, 0, displs, counts, inum, if, jf, kf)
        if (myrank == 0) then
          do k=1,kf
            do j=1,jf
              do i=1,if
                a(m) = dmax1(a(m), dabs(d(i,j,k)))
                var1 = d(i,j,k)
                if (m == 8) var1 = var1 * 1.D-20
                b(m) = b(m) + var1 * var1
              end do
            end do
          end do
        endif
      end do
      return
      end subroutine sub1
```

図 6.3.2-1 リダクション処理の変更前のコード

```
FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS V.OP  AVER.    VECTOR I-CACHE O-CACHE   BANK CONFLICT    ADB HIT  PROC.NAME
           TIME[sec](  %)    [msec]                   RATIO V.LEN      TIME    MISS    MISS CPU PORT  NETWORK   ELEM.%

【チューニング前】
     115    23.042(  2.6)   200.361   2124.4      0.0 96.12 112.8    20.591   0.005   0.020    0.538   10.424     0.21 gather_root
      10     0.628(  0.1)    62.773  60083.8  35980.1 99.24 243.5     0.628   0.000   0.000    0.000    0.123     0.03 sub1
```

図 6.3.2-2 リダクション処理の変更前の FTRACE 情報

```
【チューニング前】
      Memory Size (MB)              :       11392.031250 ランク 0 番
      Memory Size (MB)              :        3136.031250 ランク 1 番
```

図 6.3.2-3 リダクション処理の変更前の PROGINF 情報(メモリ使用量)

(2) チューニング内容

図 6.3.2-4 にチューニング後のコードを記載する. 各プロセスは自分が保持しているデータで部分和と部分最大値を計算する. 総和と最大値を求める処理は, MPI_REDUCE を使用し各プロセスで求めた部分和と部分最大値をランク 0 番に集め, ランク 0 番で実施する. 総和と最大値を求めるためにモデル全体のデータをランク 0 番に通信する必要が無くなったため格納先配列を削除し, 新たに部分和・部分最大値を格納する変数を追加する. メモリ使用量の削減とランク 0 で行っていた計算処理の分散が可能となる.

```
!チューニング後
      subroutine sub1(a, b)
      use mpi
      use mpisub
      include "subcom_g.inc"
      real*8,dimension(mf) :: a, b
      real*8,dimansion(inum,jf,kf) :: c
      real*8 :: var1
      real*8 ::wk_var1, wk_var2
C
      do m=1,mf
        wk_var1 = 0.0D0
        wk_var2 = 0.0D0
  :
（省略）
  :
        do k=1,kf
          do j=1,jf
            do i=1,inum
              wk_var1 = dmax1(wk_var1, dabs(c(i,j,k)))
              var1 = c(i,j,k)
              if (m == 8) var1 = var1 * 1.D-20
              wk_var2 = wk_var2 + var1 * var1
            end do
          end do
        end do
        call mpi_reduce(wk_var1, a(m), 1, mpi_double_precision,
     &                  mpi_max, 0, mpi_comm_world, ierror)
        call mpi_reduce(wk_var2, b(m), 1, mpi_double_precision,
     &                  mpi_sum, 0, mpi_comm_world, ierror)
      end do
      return
      end subroutine sub1
```

部分和・部分最大値を格納する作業変数を定義

部分和・部分最大値を計算

図 6.3.2-4 リダクション処理の変更による高速化修正後のコード

(3) 性能分析

図 6.3.2-5, 図 6.3.2-6 に, チューニング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	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
【チューニング前】													
115	23.042(2.6)	200.361	2124.4	0.0	96.12	112.8	20.591	0.005	0.020	0.538	10.424	0.21	gather_root
10	0.628(0.1)	62.773	60083.8	35980.1	99.24	243.5	0.628	0.000	0.000	0.000	0.123	0.03	sub1
【チューニング後】													
15	2.677(0.3)	178.454	2242.1	0.0	95.91	108.5	2.370	0.001	0.002	0.068	1.185	0.22	gather_root
10	3.116(0.4)	311.612	2526.1	164.5	98.90	205.1	2.787	0.000	0.001	0.043	0.280	0.01	sub1

図 6.3.2-5 チューニング前後の FTRACE 情報

```
【チューニング前】
      Memory Size (MB)              :       11392.031250   (ランク 0 番)
      Memory Size (MB)              :        3136.031250   (ランク 1 番)
【チューニング後】
      Memory Size (MB)              :       10816.031250   (ランク 0 番)
      Memory Size (MB)              :        2560.031250   (ランク 1 番)
```

図 6.3.2-6 チューニング前後の PROGINF 情報(メモリ使用量)

チューニング後は MPI_REDUCE に置き換えたサブルーチン GATHER_ROOT の呼び出し回数が減少しており, サブルーチン sub1 にコストが吸収されている. GATHER_ROOT と sub1 の実行時間の合計が 23.7 秒から 5.8 秒に短縮された. また, 各プロセスのメモリ使用量も約 0.5GB ずつ削減することができた.

6.3.3. 分散 I/O による I/O の効率化

(1) チューニング方針

プログラムの MPI 化に伴い，ファイル出力を各プロセスが実施するようにする(分散 I/O)ことで，ファイル出力時間を短縮することができる．本事例は，リスタートファイルや可視化ファイルなどのファイル出力について，MPI 化前には全データの出力を 1 つのプロセスで実行していたものを，MPI 化に伴い各プロセスが保持するデータを出力するようにすることでファイル出力時間の短縮を図るものである．ただし，出力したファイルを読み込む際に複数ファイルの情報を集約する処理が必要になる．

図 6.3.3-1 にチューニング前のコード(MPI 化前のコード)を記載する．リスタートファイルとして，計算途中の結果をファイル出力している．MPI 化前のプログラムでは 3 次元領域における計算領域全ての値を出力している．3 次元領域を 2 次元分割した MPI プログラムでは，各プロセスが異なる領域の計算結果を保持しているため，各プロセスが保持する値を出力するようにして MPI 化することでファイル出力処理の時間短縮が可能となる．表 6.3.3-1 にチューニング前の実行時間を記載する．

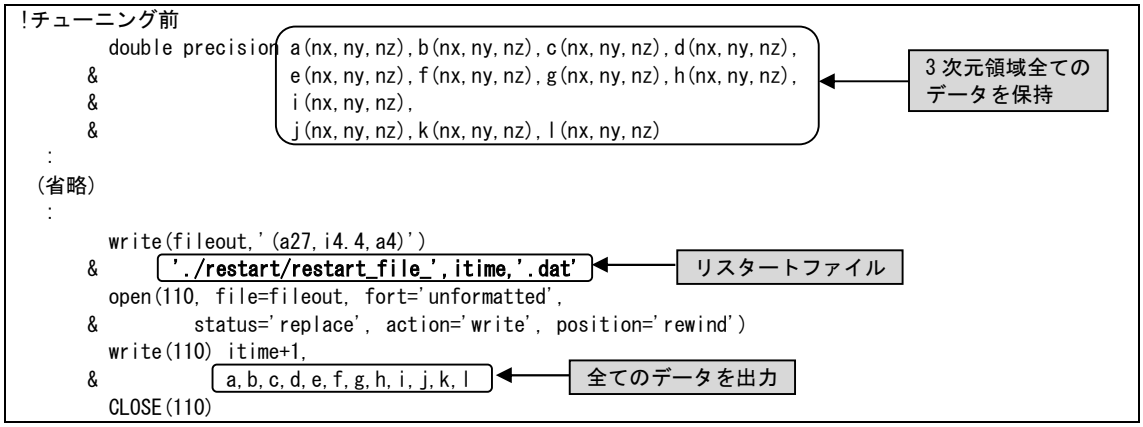


図 6.3.3-1 リスタートファイル出力処理の修正前のコード

表 6.3.3-1 チューニング前のファイル出力処理時間

	修正前
実行時間	92.4 秒

(2) チューニング内容

図 6.3.3-2 にチューニング後のコード(MPI 化後のコード)を記載する．MPI 化前のコードでは 3 次元領域全ての計算結果を出力していたが，MPI 化後のコードでは計算領域が 2 次元分割されている．1 プロセスあたりの出力対象のデータサイズは，MPI 化前のデータサイズに比べて MPI 後のデータサイズは約(1/プロセス数)となる．各プロセスが保持するデータを出力することで 1 プロセスあたりの出力データサイズが減少し，実行時間が短縮される．

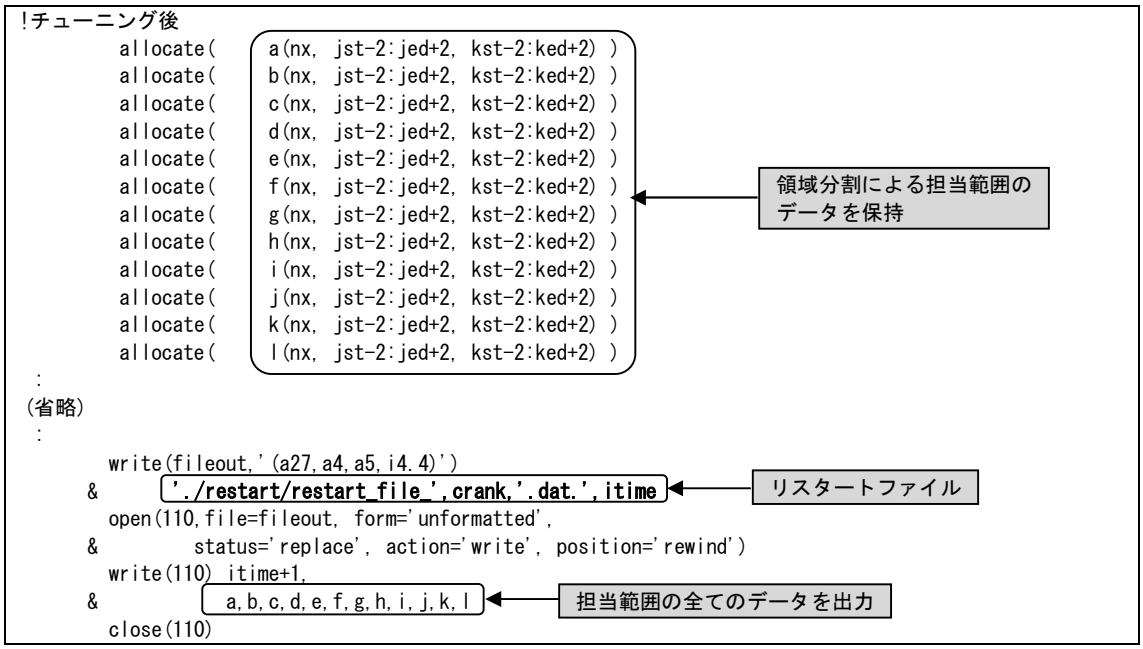


図 6.3.3-2 リスタートファイル出力処理の修正後のコード

(3) 性能分析

表 6.3.3-2 にチューニング前後の実行時間を示す.

表 6.3.3-2 チューニング前後のファイル出力処理時間

	修正前	修正後
実行時間	92.4 秒	10.1 秒

ファイル出力処理を各プロセスに分散させることにより, 実行時間が 92.4 秒から 10.1 秒に短縮されている.

6.3.4. 転置通信処理のシフト通信処理への変更による高速化

(1) チューニング方針

3 次元空間を領域分割しMPI 化を行う場合, X 軸, Y 軸, Z 軸のいずれかの軸で空間を分割し, それぞれのプロセスに分割したデータを割り当てる. プログラム内の計算処理において, 分割した空間の境界面で隣接する領域のデータを参照する場合, データの転送が必要になる. そのため, 領域分割を行う軸を, データ転送量が少なくなるように決めることが推奨される.

本事例では, プログラム内に複数の処理があり, ある処理は Z 軸で, 他のある処置は Y 軸で分割したほうがよいため, 計算の途中で分割の軸を変更する処理として MPI_Alltoall 通信を用いた転置通信処理を行っている. ただし, 転置通信処理のコストが大きいという問題があった.

図 6.3.4-1 にチューニング対象箇所のコードイメージを記載する. サブルーチン sub_xy1 は Z 軸分割で計算を行うのがよい処理である. その後のサブルーチン sub_z ではY 軸分割で計算を行うのがよい処理であるため, サブルーチン sub_z の実行前にサブルーチン sub_rotate で転置通信処理を行い, 分割の方向を Z 軸から Y 軸に変更している. その後実行されるサブルーチン

sub_xy2 では，Z 軸分割で計算を行うのがよい処理であるため，sub_xy2 の実行前に再度 sub_rotate による転置通信処理を行っている．

図 6.3.4-2 にチューニング前の MPI ランク 0 における FTRACE 情報を示す．転置通信処理を実施しているサブルーチン sub_rotate の実行時間がプログラム全体の実行時間の約 40% を占めている．そこで，サブルーチン sub_z の処理を Z 軸方向分割のまま行うことを検討し，転置通信処理の実行時間を削減する．

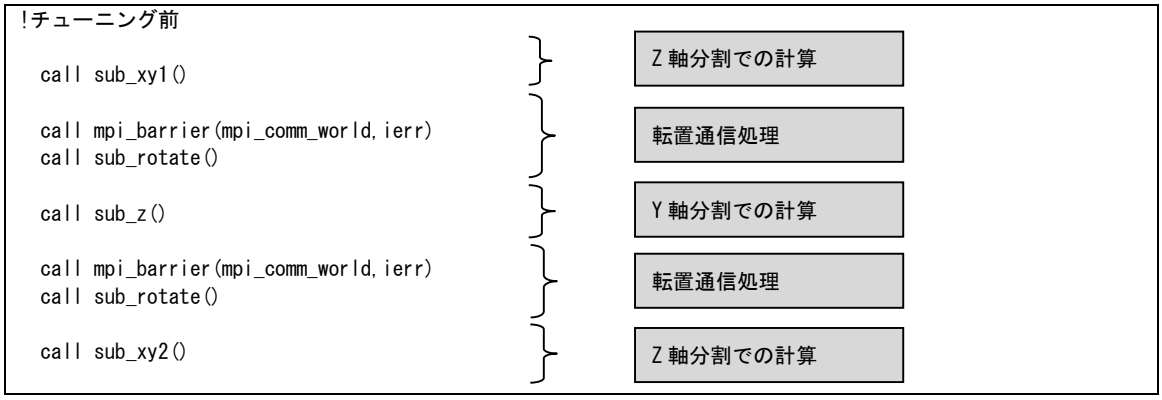


図 6.3.4-1 転置通信処理のシフト通信処理への変更前のコード

FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
【チューニング前】													
200	393.709(39.9)	1968.545	3077.2	27.0	97.82	254.3	339.383	0.055	0.401	4.850	63.872	0.13	sub_rotate
1	217.360(22.0)	217360.045	12396.1	5853.4	99.52	252.2	199.619	0.040	0.069	2.885	42.111	25.20	main_
100	9.761(1.0)	97.607	24106.6	8182.1	99.39	254.2	9.626	0.001	0.130	0.625	4.554	27.58	sub_z

図 6.3.4-2 転置通信処理のシフト通信処理への変更前の FTRACE 情報

(2) チューニング内容

サブルーチン sub_z における Z 軸方向の処理は漸化式に相当する(DO ループのある反復において i 番目の要素を更新するために，前の反復で更新された i-1 番目の要素を参照する必要がある)ため，Z 軸方向の DO ループをインデックス順に処理する必要がある．そのため，図 6.3.4-3 のように XY 平面(Y 軸方向)で領域分割した場合には単純に並列化することができる．この処理を Y 軸方向ではなく Z 軸方向で領域分割して処理するためには，Z 軸方向の計算を順番に処理する必要があるため，図 6.3.4-4 のように各ランクが自身の担当するところまで計算した後で，端の XY 平面データを隣接するランクに送付(シフト通信)し，次のランクが計算を開始するようにすればよい．

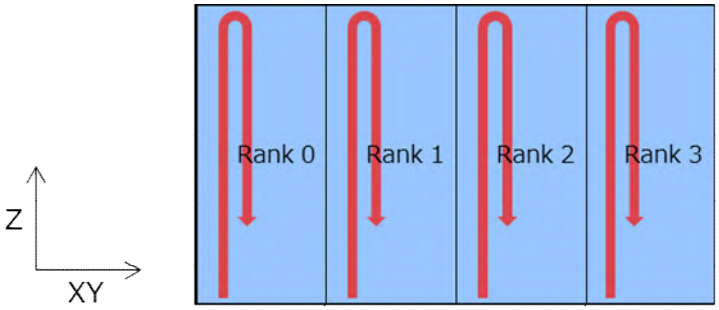


図 6.3.4-3 XY 平面で分割した場合の処理イメージ

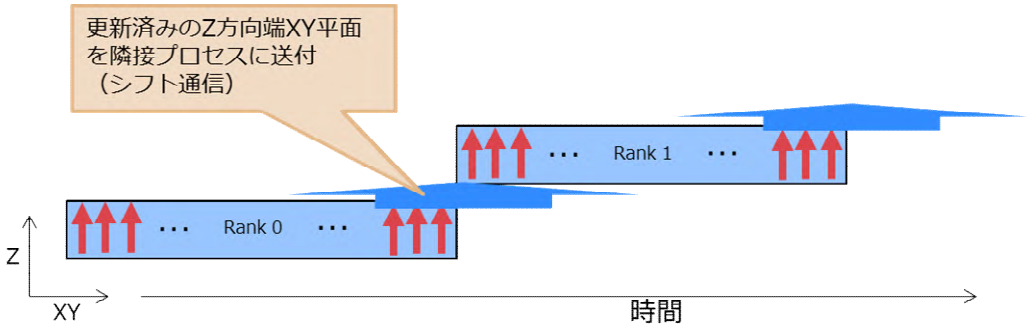


図 6.3.4-4 Z 軸方向で分割した場合の処理イメージ

ただし，この方法ではランク 0 の計算が全て終了し，隣接するランクへのシフト通信が完了してからランク 1 の処理を開始する必要があるため非並列で処理を行う必要がある．そこで，XY 軸方向を小さなブロックに分け，ブロックごとにパイプライン的に処理することで高速化を行う．図 6.3.4-5 に，ブロッキングを適用した場合の処理イメージを記載する．

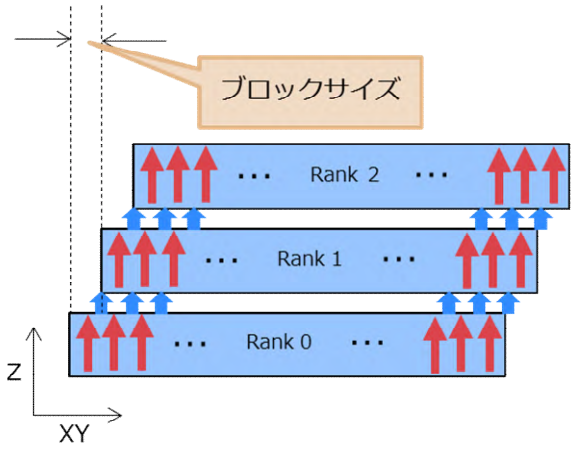


図 6.3.4-5 ブロッキングを適用した場合の処理イメージ

ブロックサイズ分の計算が終了した時点でシフト通信を行うことで，隣接するランクは処理を開始することができる．図 6.3.4-6 にチューニング後のコードを記載する．転置通信処理のシフト通信処理への変更により，サブルーチン sut_rotate と，サブルーチン sub_z が，サブルーチン

sub_shift に置き換わっている. 処理内容の詳細は割愛するが, サブルーチン sub_shift では, Z 軸方向分割のまま計算処理を行い, シフト通信を行っている.

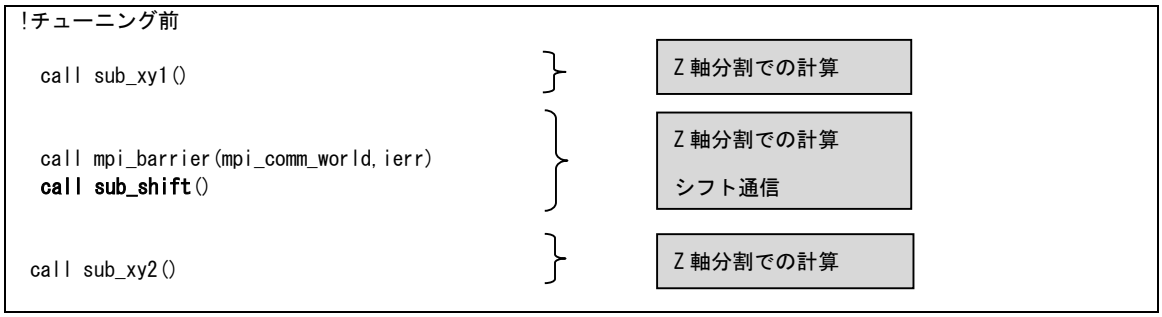


図 6.3.4-6 転置通信処理のシフト通信処理への変更後のコード

(3) 性能分析

図 6.3.4-7 にチューニング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	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	ADB HIT ELEM.%	PROC.NAME
【チューニング前】													
200	393.709(39.9)	1968.545	3077.2	27.0	97.82	254.3	339.383	0.055	0.401	4.850	63.872	0.13	sub_rotate
1	217.360(22.0)	217360.045	12396.1	5853.4	99.52	252.2	199.619	0.040	0.069	2.885	42.111	25.20	main_
100	9.761(1.0)	97.607	24106.6	8182.1	99.39	254.2	9.626	0.001	0.130	0.625	4.554	27.58	sub_z
【チューニング後】													
1	234.272(32.8)	234272.257	11555.5	5431.0	99.46	252.4	207.599	0.044	0.104	2.955	48.396	24.94	main_
100	112.250(15.7)	1122.497	5449.5	567.7	98.69	253.8	100.158	0.119	0.633	1.469	25.597	11.44	sub_shift

図 6.3.4-7 転置通信処理のシフト通信処理への変更前後の FTRACE 情報

全ての処理を Z 軸方向分割のまま行うことで, MPI_Alltoall 通信を用いた転置処理自体が不要(隣接プロセスとのシフト通信に変更)となり, さらに処理を小さなブロックに分けブロックごとにパイプライン的に処理することで高速化できる. メインプログラムの実行時間が 17 秒程度増加しているが, 通信と計算部分の実行時間は 403.5 秒から 112.3 秒に短縮されている.

7. 三次元可視化と可視化事例の紹介 SENAC Vol.49, No.3 (2016.7)より転載

情報部情報基盤課 大泉健治 小野敏 山下毅 齋藤敦子 佐々木大輔 森谷友映

7.1. はじめに

東北大学サイバーサイエンスセンター(以下, 本センター)では, 大規模科学計算システムを利用して得られたシミュレーション結果を可視化する環境として「三次元可視化システム」を提供しています. このシステムでは, GPU を搭載した 4 台の計算サーバと可視化ソフトウェアによる可視化処理や, 大画面ディスプレイを使った高精細な三次元立体視を行うことができます. また, 利用者支援活動の一環として, 本センターの技術職員による可視化支援の取り組みも行っています. 本稿では, 三次元可視化システムの概要とこれまでに本センターで支援を行った可視化事例を紹介します.

7.2. 三次元可視化システムの概要

三次元可視化システムは, 三次元立体視対応 50 インチ LED モニタを 12 面配置した大画面ディスプレイと, 演算結果の可視化処理およびディスプレイへの描画を行う可視化サーバ 4 ノードで構成されています. 大画面ディスプレイは最大 7,680×3,240 画素の高精細表示が可能です. 可視化サーバは, 各ノードにインテル Xeon プロセッサ E5-2670 を 2 基, メモリを 64GB, グラフィックボード NVIDIA Quadro K5000 を搭載しています. 可視化サーバから本センターの大規模科学計算システムのファイルサーバに直接アクセスできるようになっており, 本センターの計算機で得られたデータを別環境にコピーすることなく三次元可視化システムで利用することができます. なお, 研究室等で計算したデータを持ち込んで利用することも可能です. 可視化ソフトウェアは CYBERNET の AVS/Express MPE を採用しており, 大画面ディスプレイに可視化コンテンツを三次元表示することができます. 表示された可視化コンテンツは, 液晶シャッターメガネを通すことで立体的に見えます. 再生しながら自由自在に回転・拡大・移動することができ, 様々な視点から可視化コンテンツを見ることができます.

三次元可視化システムは, 本センター1F の可視化機器室に設置しています(図 1). 可視化機器室の壁面と大画面ディスプレイはほぼ同等の大きさのため, より没入感のある三次元立体視を体験することができます. 大画面ディスプレイは 12 面全てを使用した全画面立体視の他に, 3×3 画面, 2×2 画面など様々な表示パターンが可能で, プレゼンテーションや Polycom によるテレビ会議等でも利用することができます(図 2).

三次元可視化システムの利用方法は, 本センターのホームページ[1]をご参照ください.

図 1 三次元可視化システム(可視化機器室)

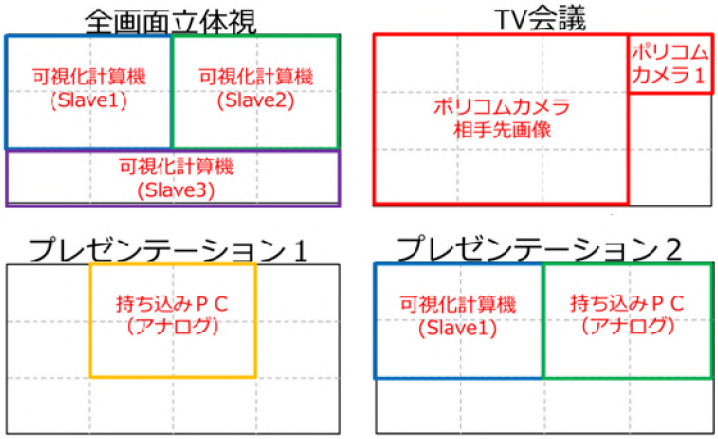


図 2 ディスプレイ表示パターンの例

図 3 三次元可視化システム利用の様子

7.3. AVS/Express による可視化コンテンツ作成

可視化コンテンツの作成には，可視化ソフトウェア AVS/Express を使用します．AVS/Express は GUI 画面上で，モジュールと呼ばれる四角い箱の形をした様々な可視化機能をつなぎあわせて可視化ネットワークを作ることで，可視化コンテンツを作成していきます(図 4)．利用可能なモジュールはおよそ 1,000 個もあり，それらを任意に組み合わせることで多様な可視化処理を行うことができます．なお，スクリプトで

可視化処理を自動化し実行することも可能です.

入力データは, テキスト形式／バイナリ形式のどちらにも対応しています. データ読込速度はバイナリ形式の方が速いので, 大規模データの場合はバイナリ形式でデータを用意することをお勧めします. また, 大規模なデータは, 読込だけでなく, 可視化処理(加工／描画／出力)にも非常に時間がかかります. そこで, 可視化処理では一般的にデータの間引きを行います. AVS/Express にはデータを間引くためのモジュールも備わっていますので, あらかじめ間引いた入力データを用意しなくとも, AVS/Express 上で可視化した画像を見ながらデータの間引き度合いを調整することが可能です. 入力データが構造格子型・離散データ・UCD 型の場合は, データのフォーマット情報を記述したヘッダファイル(AVS 共通書式)を介してデータを読み込みます. よって, 可視化用にフォーマットを整えた入力データを別に用意するのではなく, シミュレーション結果をそのまま入力データとして読み込むことができます. (ただしデータのフォーマットによっては整形が必要な場合もあります.)そのほか, plot3D や STL など多数のフォーマットに対応しています.

出来上がった可視化コンテンツは, 画像や動画として保存して持ち出すことができます. 汎用的な画像・動画の形式での保存も可能ですが, AVS/Express 独自の 3D アニメーションファイル「GFA 形式」での保存をお勧めします. GFA 形式のファイルは, AVS/Express をインストールしていないパソコンでも, CYBERNET のフリービューワ「3D AVS Player」[2] を用いて再生することができます. AVS/Express 上で再生するのと同様に, 三次元動画として再生しながら自在に視点変更することができ, プレゼンテーション等でも利用することができます.

AVS/Express は, 本センターの可視化機器室で利用することができます. 詳しい使い方は, 本センターの講習会資料[3]や AVS/Express の各種マニュアルをご参照ください. AVS/Express の各種マニュアルは可視化機器室にあります.

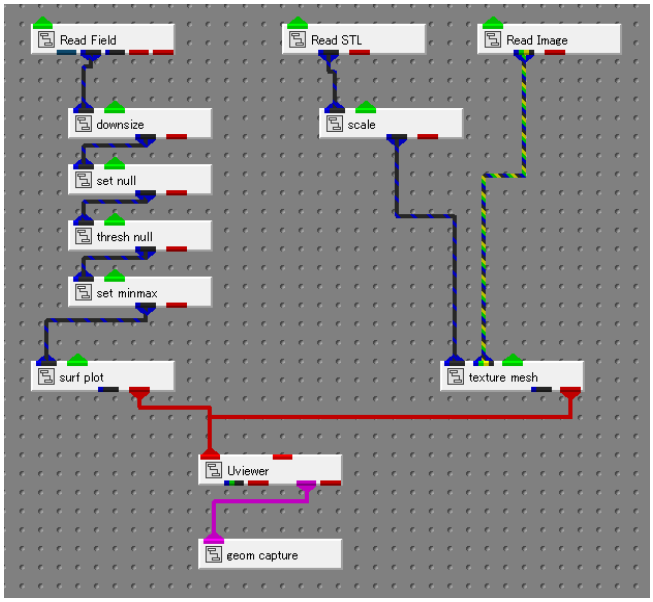


図 4 可視化ネットワークの例

7.4. 可視化事例紹介

本センターで作成および作成支援を行った可視化の事例を紹介します. これらの可視化コンテンツは, 本センターの見学コースのひとつとして, センター来訪者にも公開しています. スーパーコンピュータや並列コンピュータの計算結果を分かりやすい形で伝えられるため, 本センターや利用者の広報活動としても役立っています.

7.4.1. フラーレンの爆発解離シミュレーション

東北大学大学院理学研究科 河野研究室 山崎馨氏が研究された, X 線照射によりフラーレンが爆発解離する様子のシミュレーション[4] を三次元動画として可視化しました(図 5). 出来上がった可視化コンテンツのファイルサイズは 94MB(GFA 形式), 201 フレームからなります. 粒子の色は, 電荷の違いにより色づけしています. 作成した可視化コンテンツを, 本研究者の河野先生, 山崎氏に三次元立体視で体感してもらったところ, 奥行き情報の視覚的な認知が可能となり, 二次元画像よりも時間経過による構造の変化を詳細に観測できるので, より深く理解することができる, 構造の妥当性の直観的な検証が可能になると期待される, との感想が得られ, 三次元立体視による有意性を感じてもらうことができました.

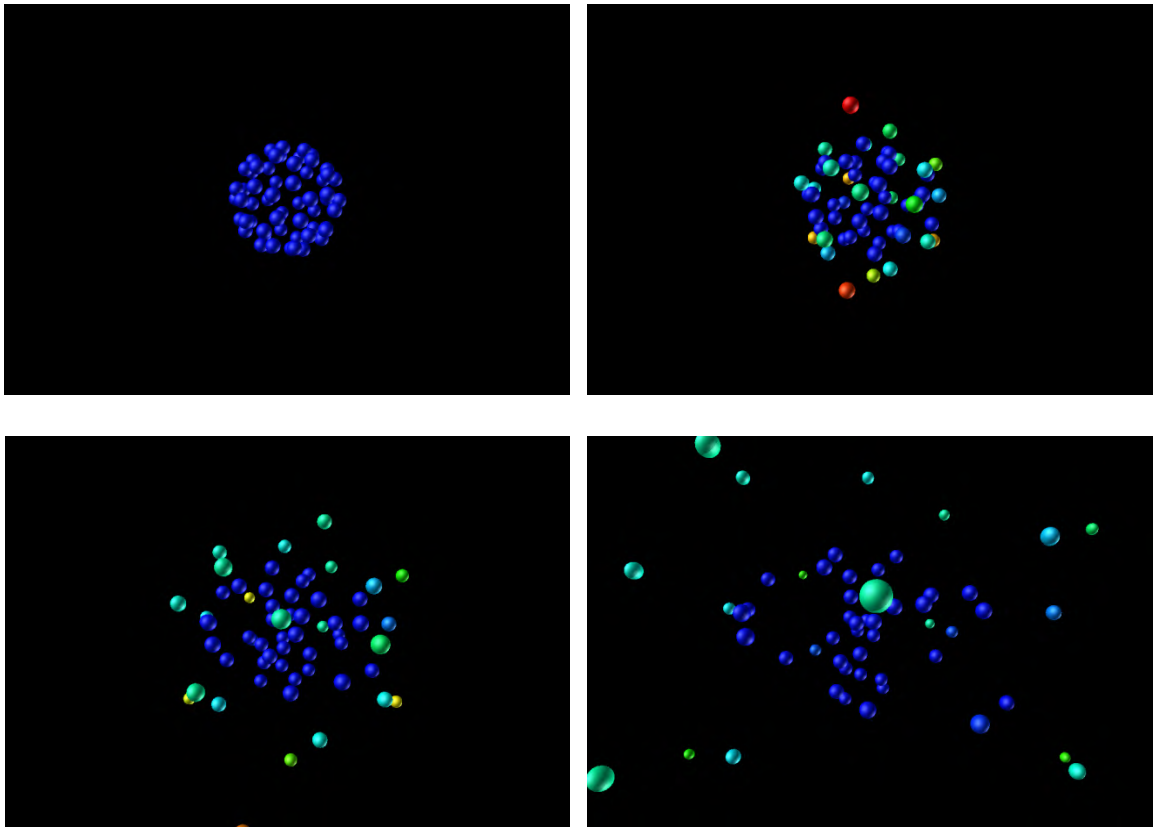


図 5 フラーレンの爆発解離シミュレーション

7.4.2. DNA 二重らせんの切断シミュレーション

東北大学大学院理学研究科 河野研究室 菱沼直樹氏が研究された, 放射線による DNA らせん構造の切断シミュレーション[5] を, 本センターの技術職員の支援のもと, 河野研究室で三次元動画として作

成されました(図 6). この可視化コンテンツは, 2015 年オープンキャンパスで「飛び出すデジタル 3D 映像を体感しよう！DNA 鎖切断動画公開」と題して公開されました. 紙面ではなかなかわかりにくい DNA のらせん構造を三次元可視化システムにより立体的に体感することができ, 見学に訪れた方々も非常に興味深く見ていました.

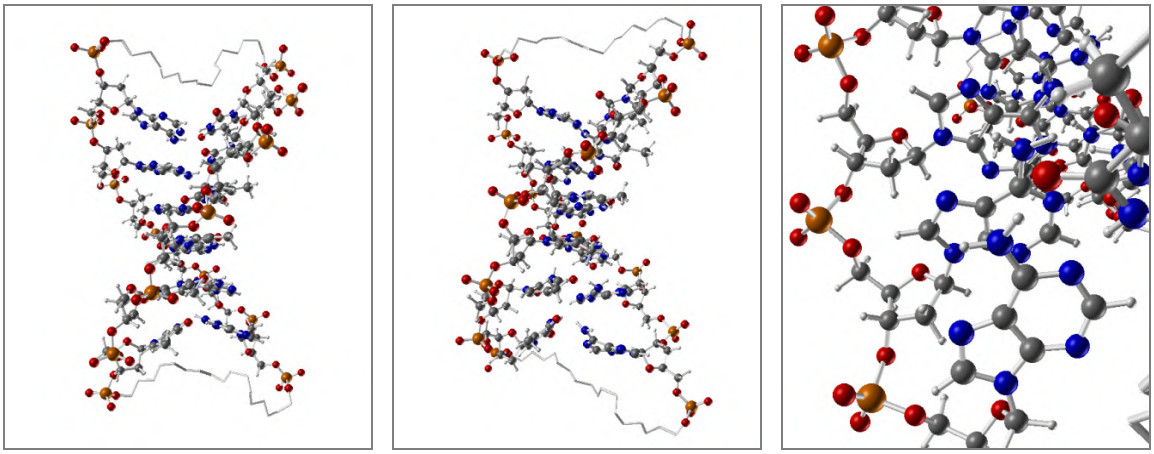
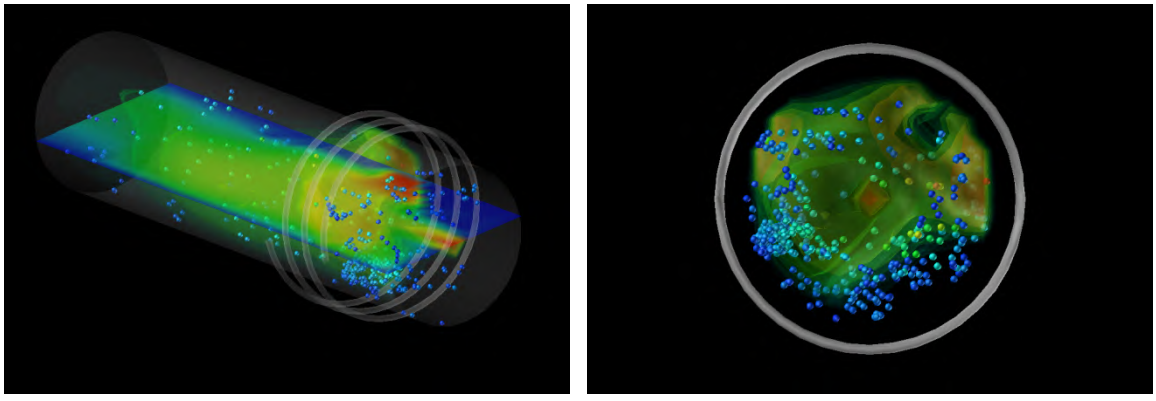


図 6 DNA 二重らせんの切断シミュレーション

7.4.3. プラズマ熱流動場のシミュレーション

大阪大学接合科学研究所 茂田正哉先生が研究された, プラズマ熱流動場のシミュレーション[6] を三次元動画として可視化しました(図 7)プラズマトーチ, RF 誘導コイル, トーチ内の温度変化, 流れ場を可視化しています. 出来上がった可視化コンテンツのファイルサイズは536MB(GFA 形式), 400 フレームからなります. 入力データは 267 万点の格子点を持つ大規模なデータであったため, 可視化するにあたり, バイナリ形式に変換およびデータの間引きを行いました. トーチ内の全体の色およびトーチ中心断面の色は, プラズマの温度変化を示しています. 流れ場は擬似的に流れに粒子を乗せて可視化しています. 粒子の色は流速で色付けをしています. 三次元立体視により, トーチ内部でらせん形状を描いて複雑に動く流れ場の様子を直感的に確認することができます.



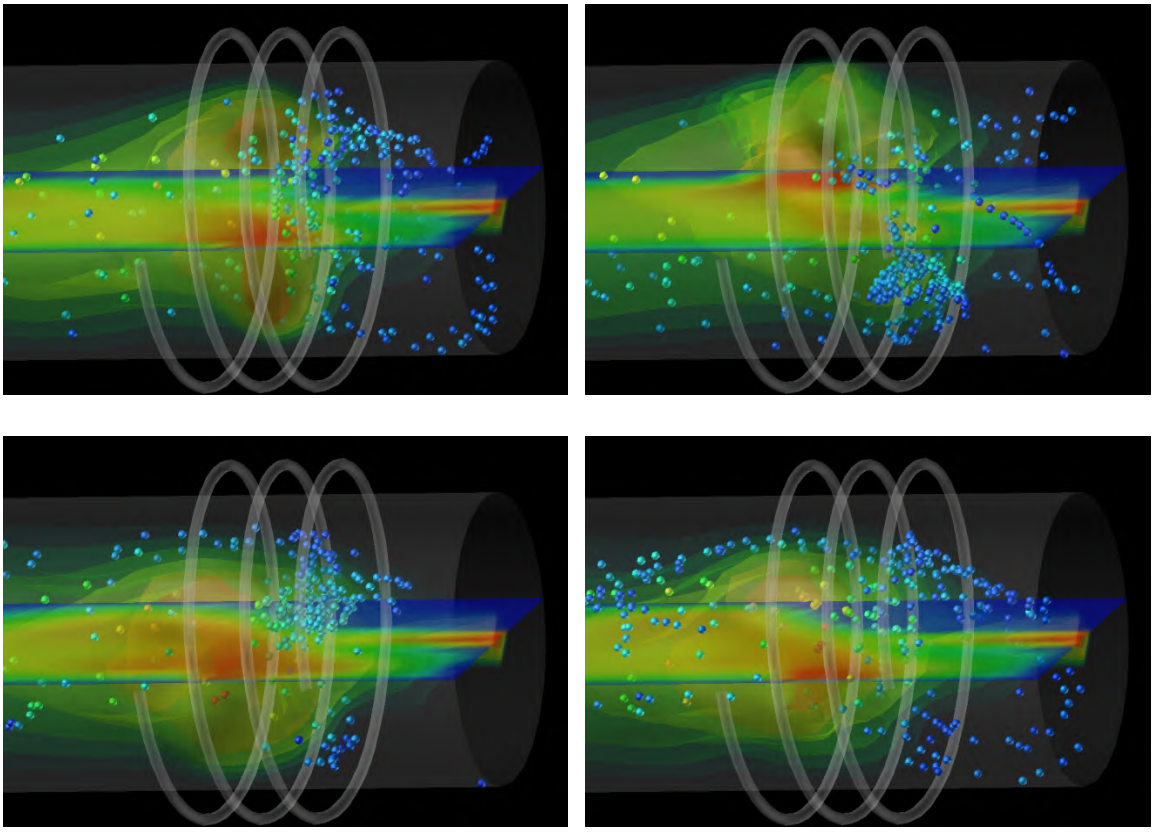
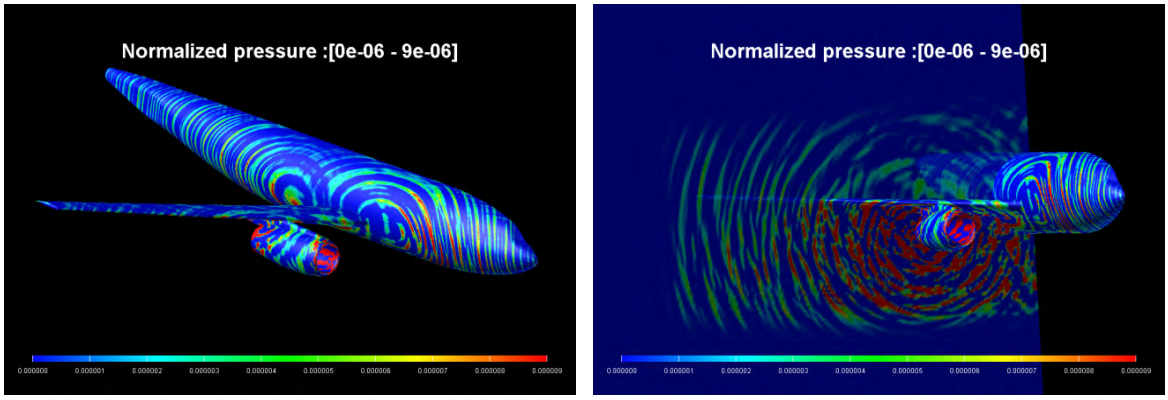


図 7 プラズマ熱流動場のシミュレーション

7.4.4. 航空機エンジン騒音の音圧伝搬シミュレーション

金沢工業大学 佐々木大輔先生，東北大学大学院工学研究科 福島裕馬氏が研究された，航空機エンジン騒音の音圧伝搬シミュレーション[7] を三次元動画として可視化しました（図 8）．ある時刻の音圧分布を様々な断面で三次元静止画にしたものをまとめて動画にしており，出来上がった可視化コンテンツのファイルサイズは 107MB（GFA 形式），25 フレームになります．赤い部分が最も音圧の高い部分を示しており，エンジン回りやエンジンに近い機体部分で圧力の高い分布になっていることが可視化した画像から見て取れます．



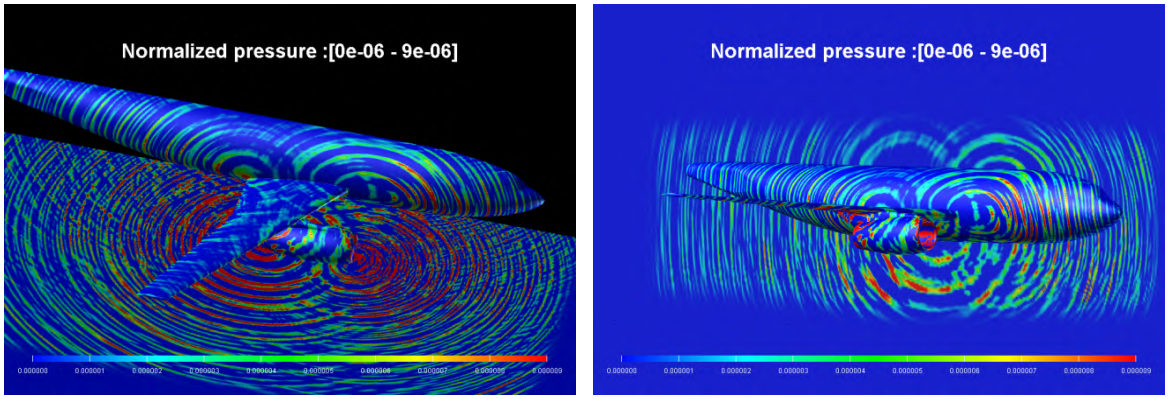


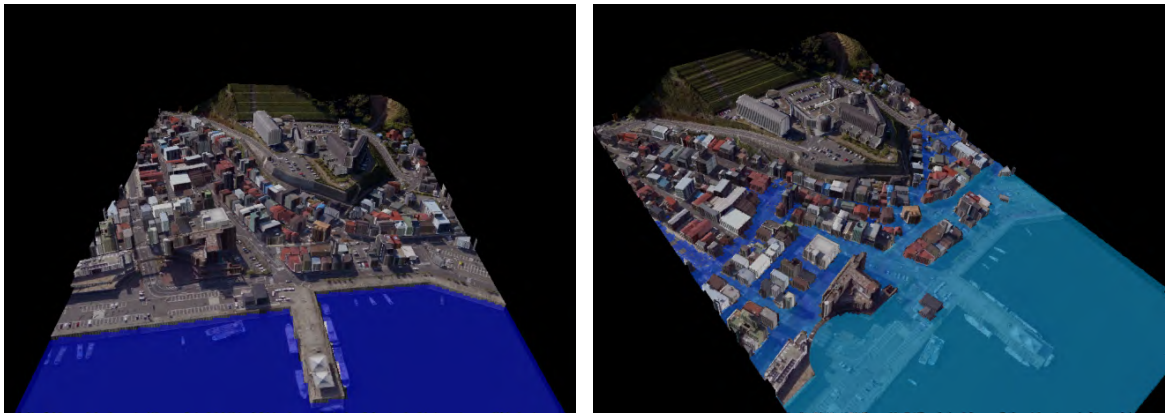
図 8 航空機エンジン騒音の音圧伝搬シミュレーション

7.4.5. 津波浸水被害の再現シミュレーション

東北大学災害科学国際研究所 越村俊一先生が研究された，東日本大震災での宮城県女川町の津波浸水被害の再現シミュレーション[8] を三次元動画として可視化しました（図 9）．500m×320m の区域を 33cm メッシュで分割して計算された大規模なデータのため，データの間引きを行い可視化しています．出来上がった可視化コンテンツのファイルサイズは 776MB（GFA 形式），1401 フレームからなります．津波の色は波高により色づけをしています．町の地形や構造物は震災前の地形データと航空写真から再現しています．津波がどのように押し寄せ，町を覆っていったのかが，可視化した動画から確認することができます．

7.5. おわりに

本センターの三次元可視化システムおよび可視化支援の例を紹介しました．本センター内で大規模科学計算からその結果の可視化までの一連の処理を行うことができます．ぜひ研究の強力なツールとして三次元可視化システムを活用していただければ幸いです．



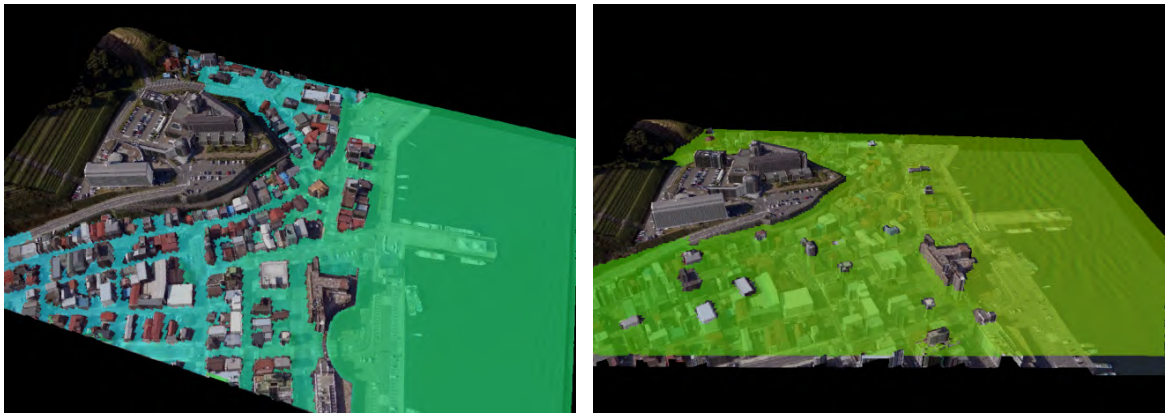


図 9 津波浸水被害の再現シミュレーション

謝辞

本稿を執筆するにあたり，可視化データを提供いただいた，東北大学大学院理学研究科 河野研究室 河野裕彦先生，山崎馨氏，菱沼直樹氏，大阪大学接合科学研究所 茂田正哉先生，金沢工業大学 佐々木大輔先生，東北大学大学院工学研究科 福島裕馬氏，東北大学災害科学国際研究所 越村俊一先生をはじめ，多くの方々にご協力ご支援をいただきました．心より感謝申し上げます．

参考文献

[1] 三次元可視化システムの利用方法 http://www.ss.cc.tohoku.ac.jp/service/vsr.html

[2] フリービューワ「3D AVS Player」 http://www.cybernet.co.jp/avs/download/player.html

[3] 講習会資料「可視化システムの利用法」 http://www.ss.cc.tohoku.ac.jp/guide/archives.html

[4] 山崎馨，上田潔，河野裕彦，「X 線自由電子レーザーパルスによるフラーレン超多価カチオン C_{60}^{q+}の爆発解離の動力学シミュレーション」, SENAC Vol.48 No.3 (2015-7) , pp.1-6 , 2015

[5] 及川啓太，菱沼直樹，菅野学，木野康志，秋山公男，河野裕彦，短鎖モデル DNA の鎖切断過程：化学反応動力学による解析，日本化学会第 96 春季年会(2016), 2016 年 3 月 24 日，同志社大学 京田辺キャンパス, 京都, 2016

[6] 茂田正哉，「DC-RF ハイブリッド熱プラズマ流の非定常 3 次元数値シミュレーション」, SENAC Vol.46 No.3 (2013-7) , pp.13-17, 2013

[7] 福島裕馬，大林茂，佐々木大輔，中橋和博，「Building-Cube Method を用いたエンジンナセルインレットからの騒音伝播解析」, SENAC Vol.47 No.1 (2014-1) , pp.35-45 , 2014

[8] S. Koshimura et al.,「The impact of the 2011 Tohoku earthquake tsunami disaster and implications to the reconstruction」, Soils and Foundations 54 (2014) , pp.560-572 , 2014

高速化推進研究活動報告　第 7 号

2018 年 12 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
http://www.cc.tohoku.ac.jp