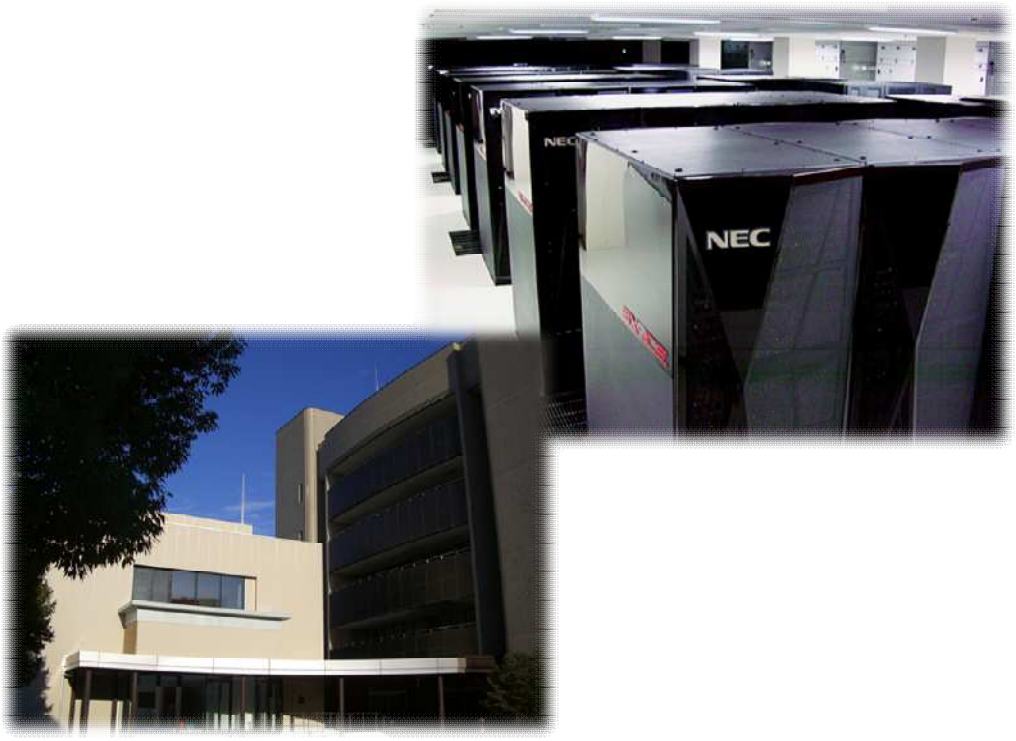
高速化推進研究活動報告

第 6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６号

目 次

1. 高速化推進研究活動報告の刊行にあたって

サイバーサイエンスセンター長　小林広明

東北大学サイバーサイエンスセンターは，その前身の大型計算機センターが 1969 年に大学の教員，その他の研究者が学術研究等のために利用する全国共同利用施設として設置されて以来，45 年が経過しました．その間，本センターは，その時代の最新のコンピュータ設備を整備し，全国の研究者に提供すると共に，その高度利用に向けて，利用者，センター教職員，ベンダー技術者が密に連携し，プログラムの高速化，および利用環境の高度化に取り組んでまいりました．特に，1997 年 9 月より，利用者，本センター，NEC の 3 者により高速化推進のための研究会を立ち上げ，高性能計算に関する共同研究を進めてきました．今回お届けする「高速化推進研究活動報告」はその成果をまとめたものであり，今回発行する第 6 号は，2011 年度から2014 年度までに得られた成果が収録されています．

サイバーサイエンスセンターの取り巻く環境はここ数年大きくかわり，これまでの学術研究者のための大規模科学計算設備共同利用施設から，

1) 2007 年：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，及びその利用支援
2) 2010 年：学際大規模情報基盤共同利用・共同研究拠点(JHPCN)」認定，および JHPCN 共同研究課題の実施
3) 2012 年：HPCI システムへの資源提供，同システム運用，および HPCI 課題の利用支援

と，その役割が広がり，センターに対する利用支援，および共同研究では質と量，両面での格段の充実が求められています．そのような中，本共同研究では，本センターの技術系の職員も中心的役割を果たしていることを特に記しておきたいと思います．

本センターの目的は，「最先端かつ世界最大級のコンピュータシステムの導入と利用環境の構築・高度化，次世代コンピュータシステムの要素技術の研究開発」ということにつきます．計算機の揺籃期においては，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，アプリケーション等を開発してきました．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，さらにわ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参りました．これらの取り組みをさらに発展させるべく，2014 年 7 月には，サイバーサイエンスセンターに HPC に関する産学連携研究拠点として「高性能計算技術開発(NEC)研究部門」を新設し，研究活動を開始しました．本研究部門は，サイバーサイエンスセンタースーパーコンピューティング部門の教員，および情報部情報基盤課技術職員に加え，NEC からアプリとシステムを専門にする２名の客員教授と客員准教授，神戸大学から１名の客員教授を迎え，センター利用者にとって真に役に立つ学術情報基盤の整備・運用・研究開発にこれまで以上に密接に連携して取り組んで行きたいと思います．

最後に，本センターとの共同研究に積極的に参加し，目覚しい成果を挙げていただいた，本センター利用者，NEC のスタッフならびに，日頃からご支援いただく文科省，および HPCI コンソーシアムのご関係各位に厚くお礼を申し上げます．本報告書を通して，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共に，皆様のプログラムの高速化の一助になれば幸甚です．

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　小松一彦　小林広明

21. はじめに

近年，コンピュータを用いたシミュレーションは多様な科学技術・学術分野おいて理論，実験と並んで重要な役割を担っている．これらのシミュレーションを支えるスーパーコンピュータは，半導体加工技術やシステムアーキテクチャの進歩に伴い，著しい進化を遂げ，高い理論演算性能を達成している．一方でメニーコア化，メモリ階層の深化等，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必要不可欠になりつつある．このような状況下でサイバーサイエンスセンターでは，スーパーコンピューティング研究部の教員，共同利用支援係，共同研究支援係の職員，導入ベンダーの計算機科学を専門とする者の知識と経験を，計算科学者である本センターの利用者と共有しながら，プログラムの高速化技術および，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．また，これらの共同研究を通して得られた知見を次期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構築に向けた研究開発も行っている．以降，本章ではサイバーサイエンスセンターの特徴的な取り組みである高速化推進研究活動について述べる．

22 大規模科学計算システム

高速化支援活動の説明に先立ち，本センターで運用している大規模科学計算システムについて述べる．本システムは，2008 年 4 月に運用開始されたベクトル型スーパーコンピュータ SX-9 (18 ノード，29.4TFlop/s，18TB) と 2014 年 4 月に導入したスカラ型並列コンピュータ LX 406Re-2 (68 ノード，31.3TFlop/s，8.5TB)から構成されている．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．

2008年から約6年半にわたり運用をしてきたSX-9は，科学技術計算の高速処理を目的に設計されたスーパーコンピュータで，スカラ型のシステムと比較して高いメモリバンド幅，コア性能を有することから，高いメモリ性能が必要となるベクトル計算や行列の計算にとりわけ優れた性能を発揮する．1 ノードあたり 16CPU，1TB の共有メモリを有する SMP 構成をとり，研究室レベルでは実行不可能な大規模シミュレーションの実行を可能にしている．また，2014年4月に導入したスカラ型並列コンピュータLX 406Re-2は，ベクトルスーパーコンピュータ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を目的としてSX-9を補完する役割を担っている．

2015 年には最新のベクトル型スーパーコンピュータ SX-ACE を導入，運用を開始する．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 707 TFlop/s，総メモリバンド幅 655TB/sec，総メモリ容量 160TB に達し，前機種では実現不可能だった超並列大規模シミュレーションを可能にしている．表 2.2-1 に現行システムであるSX-9 と次期システムであるSX-ACEの諸元を示す．コア当たりの性能，メモリバンド幅，メモリ容量はSX-9と比較して低下しているものの，マルチコアアーキテクチャ，大容量のオンチップメモリ，新たなアーキテクチャの工夫によりSX-9 を凌ぐ実効性能を実現することが期待されている．システム全体を比較すると，新システ

ムは旧システムと比較して，約 20 倍以上の理論性能，約 9 倍メモリ容量を有するシステムとなっており，大規模化の一途を辿るユーザアプリケーションの高効率実行環境を提供可能となっている．

表 2.2-1 SX-9 と SX-ACE の諸元

	性能	SX-9	SX-ACE	性能向上比
CPUあたり	コア数	1個	4個	4倍
	理論最大演算性能	118.4GFLOPS	276GFLOPS	2.3倍
	最大ベクトル演算性能	102.4GFLOPS	256GFLOPS	2.5倍
	メモリバンド幅	256GB/sec	256GB/sec	1倍
	ADB	256KB	1,024KB/コア ×4	16倍
システムあたり	CPU数	288個	2,560個	8.9倍
	理論最大演算性能	34.1TFLOPS	706.6TFLOPS	20.7倍
	最大ベクトル演算性能	29.5TFLOPS	655.4TFLOPS	22.8倍
	メモリ容量	18TB	160TB	8.9倍
サービス環境（2015年夏以降）	ノード数	64	1,024	16倍
	並列数	64	4,096	64倍
	理論最大演算性能	7.6TFLOPS	282.6TFLOPS	37倍
	最大ベクトル演算性能	6.6TFLOPS	262.1TFLOPS	40倍
	メモリ容量	4TB	64TB	16倍

図 2.2-1 に 2015 年 1 月以降の大規模科学計算システムを示す．ベクトルスーパーコンピュータ SX-ACE，スカラ型並列コンピュータ LX 406Re-2 に加え，4PB の大規模共有ディスクと三次元可視化が可能な没入型タイル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．

サイバーサイエンスセンターは，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，平成 25 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．

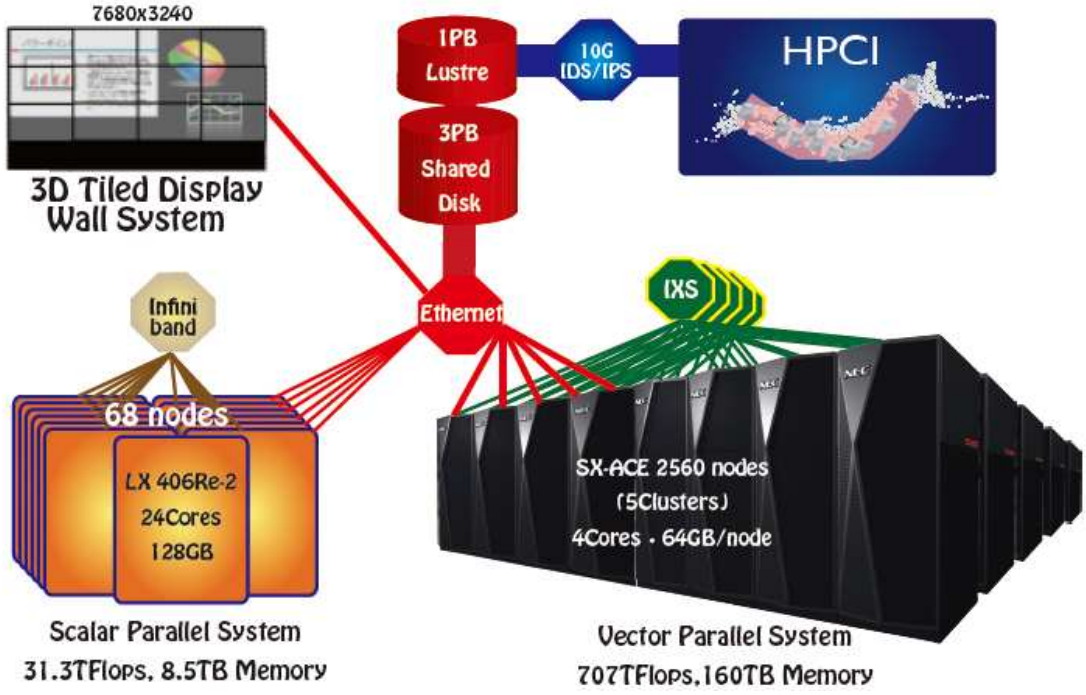


図 2.2-1 新大規模科学計算システム

2.3. 高速化推進研究活動

本センターでは 1999 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．利用者，計算機科学を専門とするセンター教職員とベンダー技術者（NEC）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いる．さらに本センターは，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，平成 22 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．平成 25 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基盤センターのスパコン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 共同研究はスパコン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だと考えられ，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．

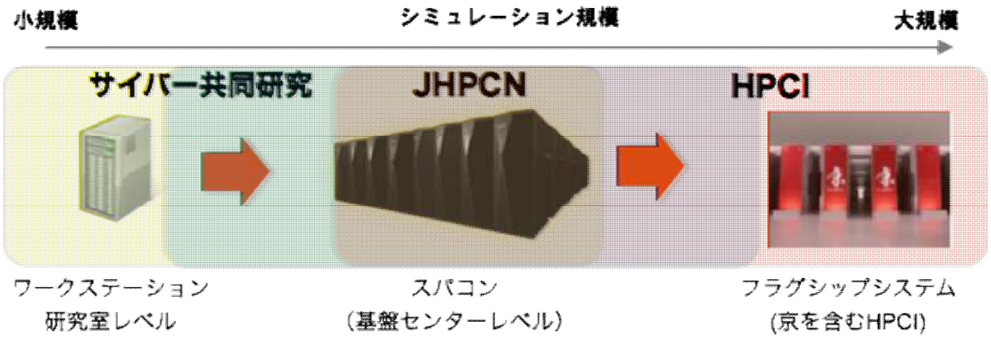


図 2.3-1 共同研究制度とシミュレーション規模

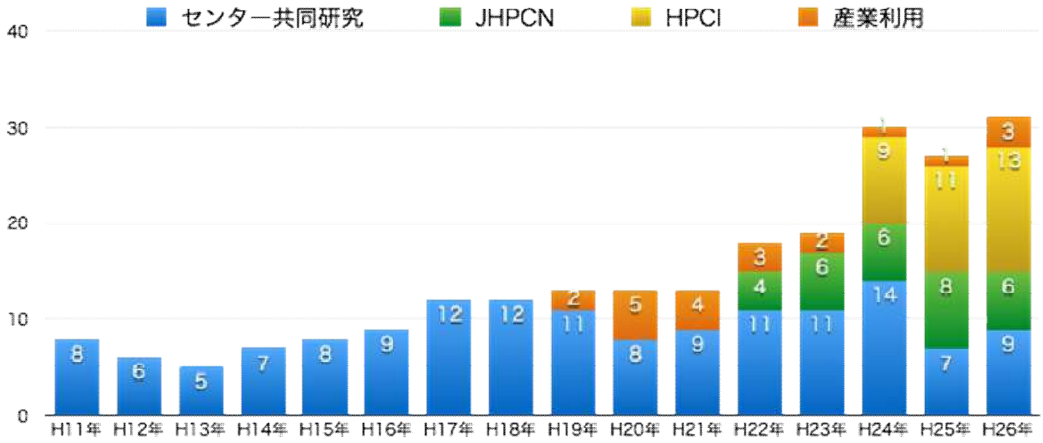


図 2.3-2 共同研究数の推移

表 2.3-1 にこれまでセンターが行ってきた高速化支援の成果を示す．平成 9 年より平成 24 年度にかけて約 170 件の共同研究に基づくコードの最適化に取り組んでおり，単体性能，並列性能ともに大幅な向上している．特に，平成 22 年度以降は 26 年度のシステム更新を見据え，これまで SMP 自動並列化を活用してきたユーザコードの MPI 並列化支援に注力してきたため，並列化性能が大幅に向上している．

次節では，これらの取り組みによって得られた知見に基づいて，我々が取り組んでいる高性能計算システム開発に関する研究開発について述べる．

表 2.3-1 高速化推進活動実績

年	H9年	H10年	H11年	H12年	H13年	H14年	H15年	H16年	H17年	H18年	H19年	H20年	H21年	H22年	H23年	H24年	H25年
件数	2	8	8	9	10	7	18	20	8	16	10	15	8	8	13	6	11
平均ベクトル化性能向上比	1.9	46.7	4.5	2.5	1.6	2.2	6.7	2.9	1.5	2.9	33	9.4	381	47	16.2	19.7	16.7
平均並列化性能向上比	11.1	18.4	31.7	8.6	4.9	2.8	18.6	4.5	4.1	8.1	1.9	5.1	3.6	48	17.2	15.3	12.9

2.4. 大規模科学計算システムの研究開発

サイバーサイエンスセンターでは，高速化支援活動をとお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，垂直積層技術を用いたマイクロプロセッサ設計，高性能低消費電力を実現するメモリサブシステムの開発，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の開発等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 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，2012 年にはマイクロプロセッサに関する国際会議でベストポスターアワードを受賞するなど国内外から高く評価されている．

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．これらの活動を活性化しながら，より実用的な研究開発に展開するべく，平成 26 年には「高性能計算技術開発(NEC)共同研究部門」を設置し，これまで以上に高速化支援に基づく積極的な研究活動を推進する環境の整備を進めている．

図 2.4-1 SC14 におけるブース展示

さらに平成 24～25 年度には，これらの研究を発展させる形で，サイバーサイエンスセンター・スーパーコンピューティング研究部が中心になり，本学災害科学国際研究所，情報科学研究科，電気通信研究所の研究者と海洋研究開発機構(JAMSTEC)，日本電気(NEC)，東京大学，大阪大学，理化学研究所，宇宙航空研究開発機構，北陸先端科学技術大学院大学，東北マイクロテックなど学外の計算科学・計算機科学の研究者・技術者の協力を得ながら，2018 年頃に実現が求められ，我が国の安全安心な社会作りと，産業界の国際競争の強化に不可欠な先端ものづくり技術の実現に資するスーパーコンピュータシステムに関するプロジェクト「高メモリバンド幅アプリケーションのための将来の HPCI システムのあり方の調査研究」を企画し，その提案が文科省の公募事業「将来の HPCI システムのあり方調査研究」の 1つ

として採択されている. 本調査研究では, 自然災害に対する防災・減災, および先進ものづくり分野での社会的・科学的課題の達成可能性を検討し, 2020 年を見据えたアプリケーションの開発ロードマップ, および将来のスーパーコンピュータに求められる性能要件を明らかにし, アプリケーション, システム, デバイスの各研究者らとともに, 図 2.4-2 に示すようなシステムの概念設計を行った.

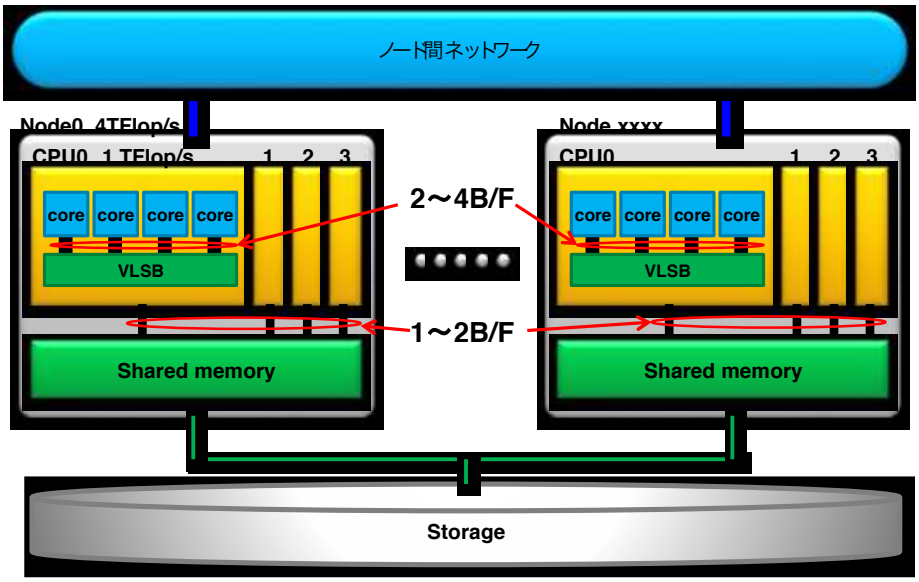


図 2.4-2 将来の HPCI システム概念設計結果

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・共同研究体制が下になっている. 高速化支援研究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 2011 年 10 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 佐々木大輔 森谷友映 齋藤敦子

本センターのスーパーコンピュータSX-9は2008年4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1ノードの全CPUを1つのジョブで占有する並列処理を運用の中心に置き，またCPU時間を制限しない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元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，2011年度から2013年度までのSX-9システム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．

3.1. 高速化推進研究活動（2011年～2013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2011年度（平成23年度），表3.1-2に2012年度（平成24年度），表3.1-3に2013年度（平成25年度）の高速化支援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 2011年度（平成23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	・スカラ変数の配列化によるベクトル化率の促進 ・ADBによるメモリアクセスの高速化		1.4倍 （6並列）
2	・ループ分割によるメモリアクセス効率の改善 ・Chopt オプションの使用	1.7倍	
3	・コード修正によるインライン展開		2.3倍 （6並列）
4	・OpenMPによる並列化 ・IF文の除去によるベクトル化率の促進	7.0倍	12.8倍 （6並列）
5	・OpenMPによる並列化		4.5倍 （32並列）
6	・MPIによる並列化		20.6倍 （32並列）

7	・ハイパープレーン法によるベクトル化率の促進 ・MPI による並列化		8.3 倍 （6 並列）
8	・MPI による並列化		16.0 倍 （27 並列）
9	・MPI による並列化		3.9 倍 （6 並列）
10	・依存関係の解消によるベクトル化率の促進 ・メモリアクセス効率の改善 ・作業配列の導入による並列化率の促進 ・並列処理オーバヘッドの削減	2.4 倍	13.4 倍 （6 並列）
11	・MPI による並列化		8.5 倍 （32 並列）
12	・ループ分割および作業配列の再利用によるメモリアクセス効率の改善 ・ADB によるメモリアクセスの高速化		2.0 倍 （6 並列）
13	・ループ展開によるベクトル化率の促進 ・演算の括り出しによる演算効率の改善	53.8 倍	

表 3.1-2 2012 年度（平成 24 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	・WRITE 文の最適化によるファイル I/O の高速化 ・コンパイルオプションによる最適化の促進		3.1 倍 （6 並列）
2	・MPI による並列化		9.6 倍※ （32 並列）
3	・MPI による並列化		17.6 倍 （32 並列）
4	・ループ展開によるベクトル化促進 ・ループ融合によるメモリアクセス回数削減	34.5 倍	
5	・MPI による並列化		30.7 倍※ （32 並列）
6	・リストベクトルの削減 ・指示行によるベクトルループ呼び出し回数削減 ・指示行によるメモリアクセス回数削減	4.9 倍	

※ I/O ルーチンを除いた部分での性能比較

表 3.1-3 2013年度　平成25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	・MPIによる並列化		3.7倍 (64並列)
2	・MPIによる並列化		33.6倍※1 (64並列)
3	・作業配列の導入によるベクトル化促進 ・ループ展開によるベクトル長の拡大	5.0倍	
4	・MPIによる並列化		※2
5	・ハイパープレーン法によるベクトル化促進 ・スカラ変数の配列化によるベクトル化促進 ・ループ移動によるベクトル長の拡大	6倍	
6	・ループ中のサブルーチンのインライン展開によるベクトル化促進 ・ループ中のエラー処理部分の移動によるベクトル化促進 ・MPIによる並列化	70倍	6.4倍 (32並列)
7	・スカラ変数の配列化によるベクトル化促進 ・スカラ変数の初期化によるベクトル化促進 ・指示行の導入によるベクトル化促進	25倍※1	
8	・ASLライブラリへの置換 ・作業配列の導入によるベクトル化促進 ・ループ中のサブルーチンのインライン展開によるベクトル化促進 ・ループ分割によるベクトル化促進 ・コンパイラによる自動並列化	9.6倍※1	8.0倍※1 (16並列)
9	・行列積内部ライブラリへの置換 ・指示行の導入によるベクトル化促進 ・ループアンローリングによるメモリアクセスの効率化	5.0倍	
10	・ループ中のサブルーチンのインライン展開によるベクトル化促進 ・ループアンローリングによるメモリアクセスの効率化	11.4倍	
11	・WRITE文の最適化によるファイルI/Oの高速化 ・コンパイルオプションによる最適化の促進 ・複素数演算方法の最適化によるメモリアクセスの効率化 ・ASLライブラリの最適化	1.4倍	

※1 I/Oルーチンを除いた部分での性能比較

※2 データによって並列数が固定されるため同じデータでの性能向上比は未測定

3.2　スーパーコンピュータSX-9のベクトル化・並列化の状況

SX-9システムの2011年度から2013年度までの3年間の利用者ジョブのベクトル化・並列化の状況を表 3.2-1に，ベクトル化率および並列化率とSX-9システムの総CPU時間に対するジョブのCPU時間の割合の関係を表3.2-2に示す．表 3.2-1はSX-9で実行した利用者ジョブをベクトル化率と並列化率とで分類し，CPU時間の割合を全CPU時間に対して百分率で表したものである．この表より，高速化推進の主要な指標であるベクトル化率と並列化率は次のような状況である．

表 3.2-1　ベクトル化率と並列化率の状況

ベクトル化率	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
90%	6	1	0	1	1	2	3	5	2	68
80%	0	0	0	0	0	0	0	0	0	2
70%	0	0	0	0	0	0	0	0	0	1
60%	0	0	0	0	0	0	0	0	0	1
50%	0	0	0	0	0	0	0	0	0	2
40%	0	0	0	0	0	0	0	0	0	1
30%	0	0	0	0	0	0	0	0	0	3
20%	0	0	0	0	0	0	0	0	0	1
10%	0	0	0	0	0	0	0	0	0	0
0%	0	0	0	0	0	0	0	0	0	0

並列化率

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	CPU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68%
ベクトル化率が90%以上のジョブ	約89%
並列化率が90%以上のジョブ	約79%

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-9の特性が十分に活かされているジョブのCPU時間の割合が高い状況は，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2008年度から2013年度までの6年間について，ジョブのCPU時間の分布を示したものである．CPU時間は各プロセスで実行されたCPU時間の合計であり，2008年度においては2000時間以上のジョブの割合が20%であったが，2013年度においては61%と過去5年間と比べてジョブの大規模化への顕著な推移が見られた．

表 3.2-3　CPU時間使用分布

CPU時間	2008年度	2009年度	2010年度	2011年度	2012年度	2013年度
0 ～ 99	18%	16%	8%	8%	8%	6%
100 ～ 999	49%	38%	39%	27%	41%	23%
1000 ～ 1999	13%	12%	13%	14%	10%	10%
2000 ～	20%	34%	40%	51%	41%	61%

3.3. 今後の取り組み

3.2 節に述べたように SX-9 の運用期間中，ジョブの大規模化・長時間化が一層進んだ．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21%あった．今後は新システムである SX-ACE 向けの高速化チューニング，及び MPI 化を含む並列処理のチューニングに力を入れ，高速化支援を行う必要があると考えている．

4. ベクトルコンピュータにおける高速化

スーパーコンピューティング研究部　小林広明　江川隆輔　小松一彦　岡部公起
情報部情報基盤課　大泉健治　小野敏　山下毅　佐々木大輔　森谷友映　齋藤敦子
日本電気株式会社　撫佐昭裕　松岡浩司　渡部修
NEC ソリューションイノベータ株式会社　曽我隆　山口健太

本章では，本センターにおいてこれまで培われてきた SX-9 のベクトル性能を向上するための手法や事例を紹介する．これらの手法や事例は今後導入されるベクトルスーパーコンピュータ SX-ACE においても有用である．

4.1. ベクトルコンピュータの特徴

近年，高性能計算機（HPC）システムを活用したシミュレーションの応用範囲は益々広がっており，同時にシミュレーションの規模も拡大し続けている．シミュレーションを行う研究者からの要求は計算規模を拡大すると同時に，計算結果を得るまでの時間は短縮したいというものである．このような研究者の要求を満たすため HPC システムの性能も年々向上し続けている．

HPC システムに採用されるプロセッサあるいはコア(以下コア)には大きく分類して，スカラ型とベクトル型がある．スカラ型コアは命令レベル並列性に基づきデータを一つずつ実行するのに対して，ベクトル型コアはデータレベルの並列性に着目して複数の演算器(パイプライン)を用いて一度に複数の処理を同時に実行する．図 4.1-1 にスカラ命令とベクトル命令の動作を示す．この図は A(I)=B(I)+C(I) と D(I)=E(I)+F(I)の２つの計算式を 100 回繰り返す DO ループの処理を表している．スカラ命令では DO 変数が１の時の A(1)とD(1)の計算結果を求め，次に DO 変数が２の時の A(2)とD(2)の計算結果を求めるという処理を，DO 変数が 100 になるまで繰り返す．それに対してベクトル命令は A(I)=B(I)+C(I)の計算を DO 変数１から100 まで一度に行い，次に D(I)=E(I)+F(I)の計算を DO 変数１から100 までを一度に行うことにより実行時間の短縮を図る．ベクトル型コアを搭載するHPC システムは，このベクトル命令の特性を活かすため，高いメモリ帯域を具備してメモリからコアへの大量のデータ供給を可能にしている．

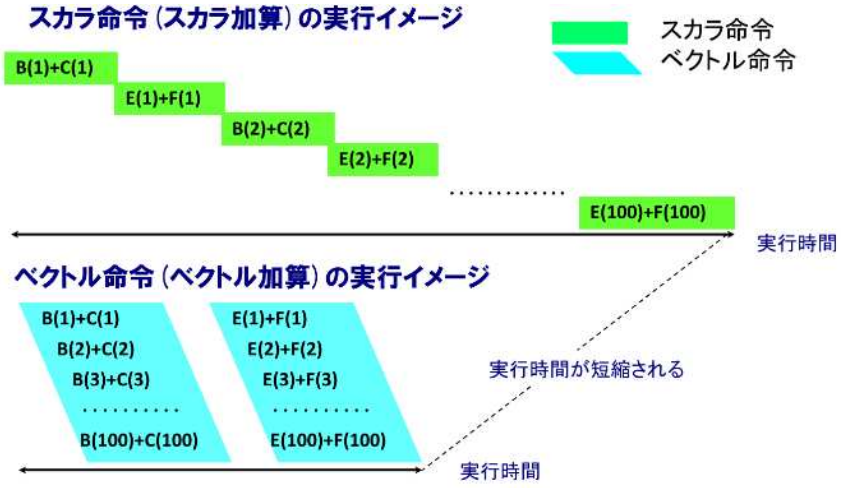


図 4.1-1 スカラ命令とベクトル命令の様子

表 4.1-1 に近年の HPC システムが搭載しているプロセッサの性能と内包するコア数，プロセッサ当たりのメモリバンド幅を示す．SX-9 以外のプロセッサでは複数の演算コアを搭載するマルチコア構成により演算性能の向上を図っているため，コア当たりのメモリバンド幅は小さくなる．しかしスカラプロセッサではメモリのバンド幅を最大限に使うためには複数のコアを使用する必要があるが，SX-ACE では 1 つコアで最大のメモリバンド幅を使うことが可能となっている．

このようにベクトルコンピュータは，高い理論演算性能とメモリ帯域を実装することで，実アプリケーションの実行において高い実効性能の達成を可能にしている．

表 4.1-1 HPC システムのプロセッサ性能とB/F 値

HPC システム名	プロセッサ			メモリバンド幅 (GB/s)
	名称	理論性能(GFlop/s)	コア数	
SX-9	—	102.4	1	256
LX406Re-2	Intel Xeon E5-269v2	230.4	12	59.7
SR16000	POWER7	245.1	8	128
FX10	SPARC64 IXfx	236.5	16	85.3
京」	SPARC64 VIIIfx	128.0	8	64
SX-ACE	—	256.0	4	256

4.2. 高速化の手法

SX システムの超高速性を十分に引き出すために重要なことは，プログラムの中でベクトル命令によって処理される部分の比率，すなわちベクトル化率と，生成されるベクトル命令の効率を可能な限り高めることである．そのためには，プログラムの最適化状況の把握・分析を行い，主要な性能指標に基づき，高速化を行う必要がある．ここでは，そのためのツール類及び，それらの性能指標と各指標に沿った高速化の取り組みかたについて述べる．

4.2.1. 性能解析ツール

(1) 編集リスト

編集リストはベクトル化および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されたかを確認することができる．編集リストはコンパイラオプション「R5」を指定することにより"入力ファイル名.L"という名前で出力される．

```
!編集リスト
FILE NAME: t5.f
PROGRAM NAME: sub
FORMAT LIST
 LINE     LOOP      FORTRAN STATEMENT
    1:                 subroutine sub(a,b,c,d,z,ix)
    2:                 real,dimension(100)::a,b,c,d,x,y,z
    3:                 integer ix(100)
    4: V-----> 1       do i = 1,100
    5: |       I           call sub2(x,a,b,i)
    6: |                   y(i) = c(i) + d(i)
    7: |       S           z(ix(i)) = z(ix(i)) + x(i) + y(i)
    8: V-----          enddo
    9:                 end
```

図 4.2-1 編集リストの例

主なループ情報，スカラ情報，手続インライン情報など編集リストの出力イメージを次に示す．

① ループ全体がベクトル化される場合

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 4.2-2 ベクトル化された DO ループの編集リスト

② ベクトル化されない場合

ベクトル化されないループには"+"が表示される．

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

図 4.2-3 ベクトル化されない DO ループの編集リスト

③　部分ベクトル化の場合

ベクトル化不可の処理がある行には "S"が表示される.

※部分ベクトルとは，ループにベクトル化を阻害する部分が含まれている場合，その前後で自動的にループを分割し，ベクトル化可能な部分だけをベクトル化する拡張機能である.

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

図 4.2-4 部分ベクトル化の編集リスト

④　配列式をベクトル化した場合(1)

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(90),b(90),c
V====== a(1:90)=b(1:90)+c
```

図 4.2-5 ベクトル化された配列式の編集リスト

⑤　配列式をベクトル化した場合(2)

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.

※ループ融合とは，繰返し数が等しい DO ループまたは配列式が複数個，連続して存在している場合一つのループに融合することである. ただし，各ループで使われているデータの定義・参照関係に，融合によって矛盾が生じる場合は行わない.

```
        real a(90),b(90),c
     integer d(90),e(90)
V-----> a(1:90)=b(1:90)+c
|       d(1:90)=int(a(1:90))
V-----  e(1:90)=d(1:90)+1
```

図 4.2-6 ループ融合とベクトル化された配列式の編集リスト

⑥　手続呼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がある行には "I"が表示される.

```
      I call sub2(x,a,b,c,I)
```

図 4.2-7 インライン展開の編集リスト

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 4.2-8 多重 DO ループの一重化の編集リスト

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 4.2-9 DO ループの入れ換えの編集リスト

⑨ ADB にデータがバッファリングされる場合

ADB を使用したベクトルロード/ストアがある行に文字 "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 4.2-10 ADB の使用の編集リスト

⑩ ベクトル化と並列化される場合

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
¦         c(i) = c(i) + a(i) * b(i)
Y-----  end do
```

図 4.2-11 ベクトル化と並列化された DO ループの編集リスト

⑪ 並列化される場合

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
¦¦          d(i,j) = 0.0d0
¦V----    end do
P-----  end do
```

図 4.2-12 並列化された DO ループの編集リスト

⑫ 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 4.2-13 複数の情報がある場合の編集リスト

(2) 簡易性能解析機能(FTRACE)

簡易性能解析情報はコンパイラオプションに「ftrace」を指定することで, 手続 (サブルーチンや関数) ごとの性能解析情報を採取することができ, プログラム実行後に解析情報が標準出力ファイルに出力される. 特に, チューニング対象とする手続を選択する際に活用すると良い. 図 4.2-14 に FTRACE 機能による解析リストの例を示す. FTRACE では実効性能と平均ベクトル長などの情報を手続 (サブルーチンや関数) 単位で取得することができる.

①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩	⑪	⑫	
PROC.NAME	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
subc	100	7.594(42.2)	75.936	79426.5	52676.9	99.48	256.0	7.594	0.0000	0.0000	0.032	0.000
subb	100	5.227(29.1)	52.272	77121.3	38261.9	99.22	256.0	5.227	0.0000	0.0000	0.000	0.000
suba	100	5.173(28.7)	51.728	58600.1	1933.29	98.97	256.0	5.173	0.0000	0.0000	0.000	0.000
test	1	0.000(0.0)	0.518	772.0	0.0	0.00	0.0	0.000	0.0000	0.0000	0.000	0.000
total	301	17.994(100.0)	59.780	7276.83	38902.9	99.28	256.0	17.993	0.0001	0.0001	0.032	0.000

図 4.2-14 解析リストの出力例

表示される各項目の意味は以下のとおりである.

①PROC.NAME	:手続名
②FREQUENCY	:手続の呼び出し回数
③EXCLUSIVE TIME	:手続の実行に要した占有の CPU 時間（秒）と，手続全体の実行に要した CPU 時間に対する比率
④AVER.TIME	:手続の 1 回の実行に要した平均 CPU 時間（ミリ秒）
⑤MOPS	:1 秒間に実行された演算数を 100 万単位で示した値
⑥MFLOPS	: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
⑦V.OP RATIO	:ベクトル演算率
⑧AVER V.LEN	:平均ベクトル長
⑨VECTOR TIME	:ベクトル命令実行時間（秒）
⑩I-CACHE MISS	:命令キャッシュミスにより発生した合計時間（秒）
⑪O-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）
⑫BANK 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）

4.2.2 ベクトル化率の向上

(1) ベクトル化率

ベクトル型スーパーコンピュータではプログラム中のベクトル処理可能な部分を高速に実行している. 図 4.2-15 に同一のプログラムをスカラ処理する場合とベクトル処理する場合の実行時間に関する概念図を示す.

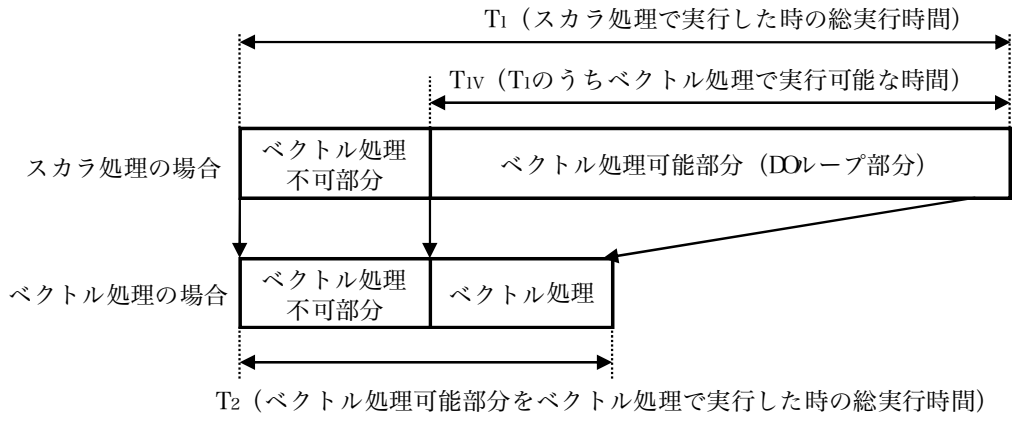


図 4.2-15 ベクトル処理による実行時間短縮のイメージ

ここで，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を T_1，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を T_{IV} とすると，ベクトル化率 α は以下の式で定義される．

$$\text{ベクトル化率}\quad \alpha = \frac{T_{IV}}{T_1}$$

また，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，

$$\text{性能向上比}\quad P = \frac{1}{(1-\alpha)+\frac{\alpha}{\beta}}$$

と表される．

この式から，ベクトル化率と性能向上比の関係は図 4.2-16 のようになる（アムダールの法則）．この図からベクトル化率 80%程度では大きな性能向上は見られず，ベクトル化率 90%を超えたあたりから急激に性能が向上していることがわかる．ベクトル型スーパーコンピュータで高い実効性能を得るためにはベクトル化率を 100%にできる限り近付ける必要がある．

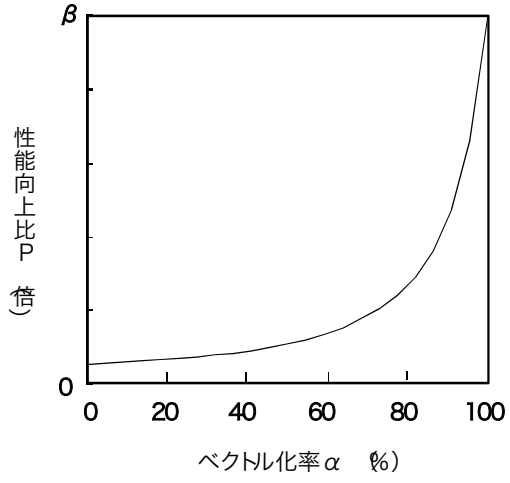


図 4.2-16 ベクトル化率と性能向上比（アムダールの法則）

ベクトル化率を求めるため，プログラムをスカラ実行する場合とベクトル実行する場合のそれぞれの実行時間が必要である．しかし，これらの実行時間は同時に得られないのでベクトル化率の算出は困難である．そのため SX-9 ではベクトル化率の代わりにベクトル演算率をベクトル化の指標としている．FTRACE などの性能解析機能に出力される情報はベクトル演算率であり，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演算命令で処理される演算要素数の割合で算出し，ほぼベクトル化率とみなせる指標であるため，後述のベクトル化率はベクトル演算率のこととする．

(2) ベクトル化を阻害する要因

ベクトル化を阻害する要因を以下に示す．

① 依存関係の有無が判断できない
※前の繰り返しで定義した値を参照する場合や間接参照

② I/O 処理(OPEN/CLOSE/READ/WRITE 文)

③ ユーザ関数(SUBROUTINE，FUNCTION)の呼び出し

④ ループの繰り返し数がループ本体の実行前に決定しない

⑤ ループの入口および出口が一つではない

⑥ ベクトル化対象外の精度を使用している
※2 バイト整数型，4 倍精度実数型，4 倍精度複素数型，1 バイト論理型，文字型,構造型はベクトル化対象外の精度

4.2.3. 平均ベクトル長の拡大

(1) 平均ベクトル長

ベクトル処理を効率的に行う上で重要な指標にベクトル長がある．ベクトル長はベクトル化対象の DO ループの繰り返し回数のことであり，効率良くベクトル処理を行うためには，できるだけループ長を長くする必要がある．

一般に命令を発行してから結果が返ってくるまでに遅延時間が存在する．この遅延時間を立ち上がり時間と呼ぶ．図 4.2-17 にベクトル処理における立ち上がり時間および演算時間の概念図を示す．網掛け部分は演算器が稼働する時間を示しており，演算を行うレジスタのデータを読み込み，演算結果をレジスタに格納するまでの時間となる．図 4.2-18 にベクトル長と立ち上がり時間の関係を示す．立ち上がり時間はベクトル長が異なる場合でも一定である．よって総演算量が等しい場合，短ベクトル長処理を繰り返すよりも長ベクトル長処理を一括して行う方が実行時間を短くすることができる．このように高速化を図るためには，DO ループのベクトル長を十分に確保し，演算時間に対する立ち上がり時間の占める割合を低くすることが重要である．図 4.2-19 にベクトル長と立ち上がり時間の関係を示す．ベクトル処理の立ち上り時間が発生するため，ベクトル長が交差ループ長より短い(4以下)場合はベクトル化を行わない方が良い場合がある．交差ベクトル長とは，ベクトル化した場合とベクトル化しない場合とで実行時間が等しくなるループ長のことである．

図 4.2-17 立ち上がり時間

図 4.2-18 ベクトル長と立ち上がり時間

図 4.2-19 ベクトル長と立ち上がり時間の関係

4.2.4. メモリアクセスの効率化

(1) メモリ競合

SX-9 ではマルチメモリバンクシステムを採用しており，メモリを 32 のバンクグループに分割している．各バンクグループには 2 要素(16 バイト)ずつ配列データが格納される．

図 4.2-20 は連続アクセスとなる DO ループの例，図 4.2-21 は格納される配列データ(二次元の配列サイズが(128,64)のとき)の概念図を示しており，CPU からメモリにアクセスする際，CPU 内の全ポートを使用して効率的にメモリアクセスを行う．

```
!連続アクセス
do j=1,ny
do i=1,nx
  :
  Rs(i,j)=Rs(i,j)+Cf(i,j)
end do
end do
```

図 4.2-20 連続アクセスの DO ループの例

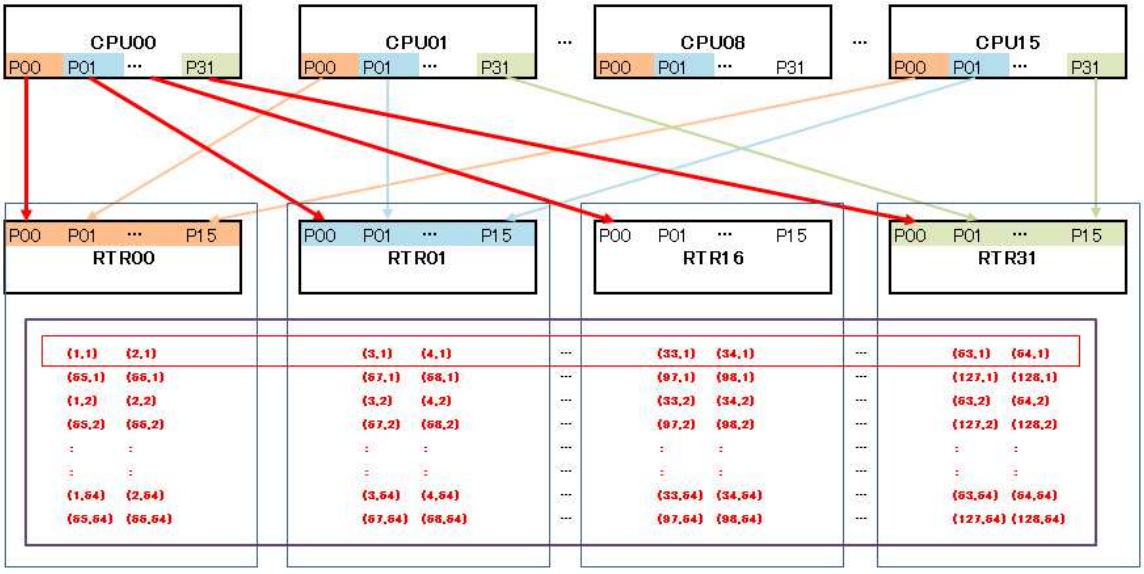


図 4.2-21 メモリバンクに格納される配列データの概念図(連続アクセス)

一方，図 4.2-22 はストライドアクセスとなる DO ループの例，図 4.2-23 は格納される配列データ(二次元の配列サイズが(128,64)のとき)の概念図を示しており，CPU からメモリをアクセスする際，アクセスするバンクグループが限定され CPU 内の特定のポート(00, 16)だけを使用してメモリアクセスを行うことになる．このように CPU 内における同一のポートにロード・ストアが集中した時に発生する CPU ポート競合や，同一のメモリバンクへのアクセスなどで発生するメモリネットワーク競合はメモリアクセスが遅延するため高速化の妨げとなる．

```
!ストライドアクセス
 do j=1, ny
 do i=1, nx, 32
   :
   Rs(i,j)=Rs(i,j)+Cf(i,j)
 end do
 end do
```

図 4.2-22 ストライドアクセスの DO ループの例

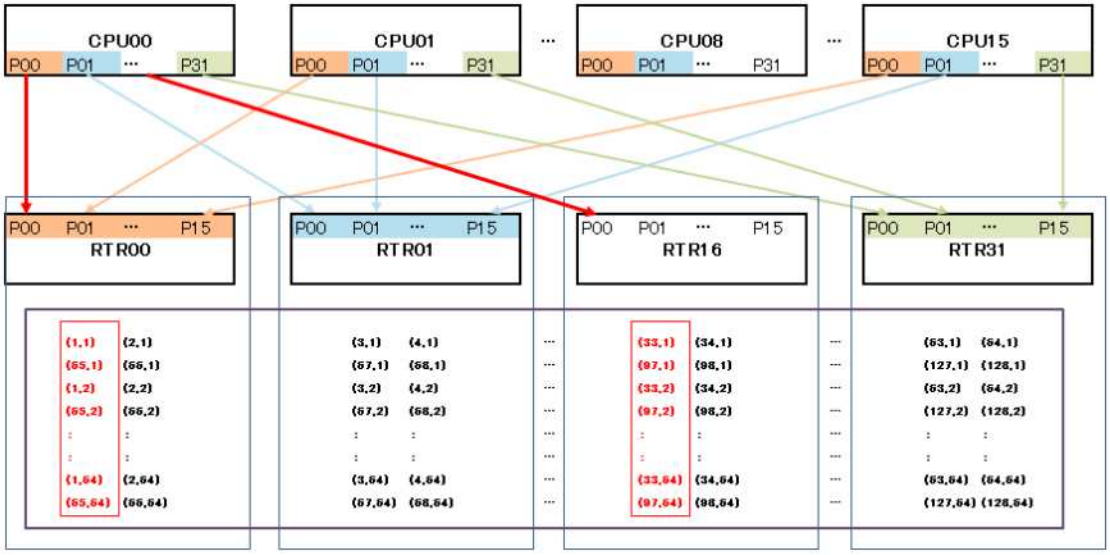


図 4.2-23 メモリバンクに格納される配列データのイメージ(ストライドアクセス)

以降にメモリアクセス性能を改善するための指針として，メモリアクセスパターンの改善と ADB の活用について述べる.

(2) メモリアクセスパターンの改善

図 4.2-24 にメモリへアクセスパターンの種類を示す．アクセスパターンの種類としては，連続アクセス，ストライドアクセス，間接アクセスの 3 種類に分けることができる.

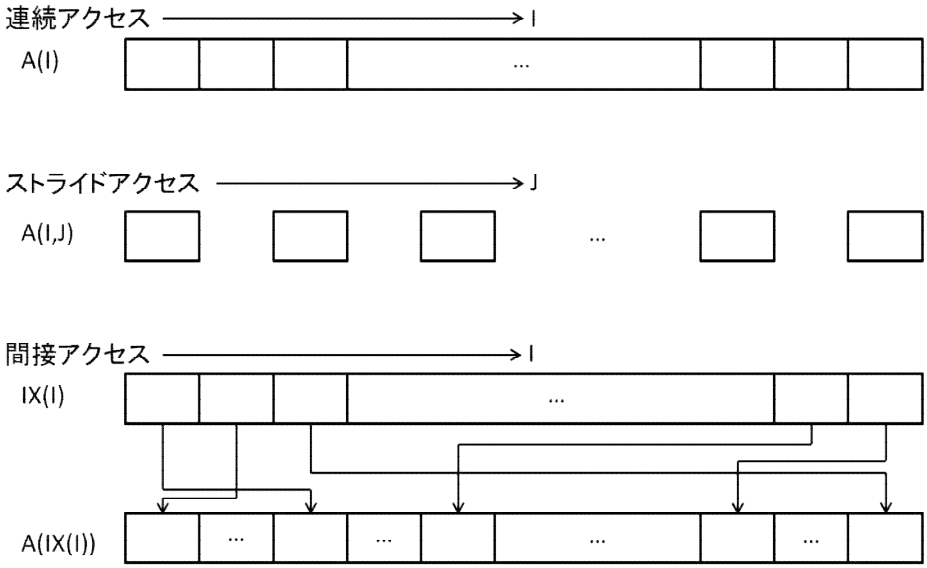


図 4.2-24 メモリのアクセスパターンの種類

図 4.2-21 に示すように，連続アクセスの場合が最も効率的なメモリアクセスを行うことができる．ストライドアクセスの場合には，図 4.2-23 に示すように，使用できるバンク数が限られてくる．この場合，ループ入れ換えなどにより，なるべく最初の次元でメモリアクセスを行うようにする．その際，ループインデックスが繰り返しごとに 1 ずつ増加あるいは減少するようにできでば連続アクセスとなり，より効率的なメモリアクセスが可能となる．間接アクセスの場合，間接アクセスとなる配列(図 4.2-24 の間接アクセスにおける配列 A)が何度も参照されるならば，最初に，この配列のデータを作業用の配列にコピーしておき，以降のループでは作業用の配列をアクセスするようにすることにより，連続アクセスにすることができる．

(3) ADB の活用

SX-9 では CPU-主記憶装置間に ADB(Assinable Data Buffer)と呼ばれるベクトルデータを選択的にバッファする機能を有している．この ADB を活用することによって，メモリアクセス性能を改善できる．

コンパイラは，再利用性があると判断した配列を自動的に ADB にバッファリングするが，コンパイラが判断できなかった配列については，ON_ADB 指示行を用いることによって，明示的に該当配列を ADB にバッファリングすることを指示することができる．例えば，間接アクセスにおいて，間接アクセスとなる配列(図 4.2-24 の間接アクセスにおける配列 A)に対し ON_ADB 指示行を用いて ADB にバッファリングしておくことにより，この配列のロードを高速に行えるようになる．

なお，ADB の容量は限られているため，ADB にバッファリングするデータのサイズに注意し，再利用性のあるデータを ADB にバッファリングするように指定することが重要である．

4.3. ベクトル化率向上の事例

4.3.1. ループ内の WRITE 文の括り出し

(1) チューニング方針

SX-9では, DO ループの一部にベクトル化対象外の文が存在すると, ベクトル化されない. もしくは, 部分ベクトル化となり, 完全にはベクトル化されない. 図 4.3-1 にチューニング前のコードを示す. このコードでは, DO ループ内にエラー検出時に実行される WRITE 文が含まれている. 入出力文はベクトル化対象外の処理であるため, DO ループ全体がベクトル化されていない. 図 4.3-2 のチューニング前の FTRACE 情報を確認すると, ベクトル化率が 0.0%であり, 全くベクトル化されていないことが分かる. そこで, ベクトル化阻害要因である WRITE 文を DO ループの外に移動させることにより, DO ループをベクトル化可能にする.

```
!チューニング前
12: +-----> DO J=2, JF
13: |+----> DO I=2, IF
 :
18: ||       IF (HZ(I,J) .GE. -30.0) THEN
19: ||         ZZ = Z(I,J,1) - RX*(M(I,J,1)-M(I-1,J,1))
20: ||       &               - RY*(N(I,J,1)-N(I,J-1,1))
21: ||         IF (ABS(ZZ) .LT. GX) ZZ = 0.0
22: ||         DD = ZZ + HZ(I,J)
23: ||         IF (DD .LT. GX) DD = 0.0
24: ||         DZ(I,J,2) = DD
25: ||         Z(I,J,2) = DD - HZ(I,J)
28: ||         IF(ABS(Z(I,J,2)).GT.100.0) THEN
29: ||         NG=1
30: ||         WRITE(*,'(A24,3I6)')'Over flow Z at (K,I,J) :', KK,I,J
31: ||         WRITE(*,*)'Within Region :', NREG
32: ||         WRITE(*,*)'Computation is unstable.'
34: ||         END IF
36: ||       END IF
37: |+-----  END DO
38: +------ END DO
```

図 4.3-1 WRITE 文括り出し前のコード

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	PROC.NAME
40	6.863(3.1)	171.564	558.3	195.4	0.00	0.0	0.000	0.000	2.239	0.000	0.000	【チューニング前】

図 4.3-2 WRITE 文括り出し前の FTRACE 情報

(2) チューニング内容

図 4.3-3にチューニング後のコードを示す. チューニング前のDOループではエラー検出時に WRITE 文によるエラー情報の出力を行っているが, 本修正においては, DO ループ内ではエラー検出時にフラグ検出用配列にフラグをセットするようにした. そして, WRITE 文によるエラー情報の出力処理は DO ループ外に移動させ, 出力が必要な場合のみ処理を行うようにした. これにより, DO ループ内にはベクトル化対象外の文がなくなるため, ベクトル化が可能となる.

```
!チューニング後
! フラグの初期化
23:                     IFLG1 = IF * JF + 1
24: VV=====             IDX = IFLG1
 :
! メイン処理
!   エラーの検出のみ実施して，エラー出力処理をループの外に出す
27: +----->             DO J=2, JF
28: |V----->               DO I=2, IF
 :
33: |||                     IF (HZ(I,J) .GE. -30.0) THEN
34: |||                       ZZ = Z(I,J,1) - RX*(M(I,J,1)-M(I-1,J,1))
35: |||                &                     - RY*(N(I,J,1)-N(I,J-1,1))
36: |||                       IF (ABS(ZZ) .LT. GX) ZZ = 0.0
37: |||                       DD = ZZ + HZ(I,J)
38: |||
39: |||                       IF (DD .LT. GX) DD = 0.0
40: |||                       DZ(I,J,2) = DD
41: |||                       Z(I,J,2)  = DD - HZ(I,J)
 :
45: |||                       IF(ABS(Z(I,J,2)).GT.100.0) THEN
53: |||                         IDX(I,J) = I + (J - 1) * IF
54: |||                       END IF
56: |||                     END IF
57: |V-----               END DO
58: +-----              END DO
 :
! エラー出力処理
61: +----->             DO J=2, JF
62: |V----->               DO I=2, IF
63: |||                       IFLG1 = MIN(IDX(I,J), IFLG1)
64: |V-----                 ENDDO
65: +-----              ENDDO
66:                    IF ( IFLG1 .LT. IF * JF + 1 ) THEN
67:                      INDX_J = (IFLG1 - 1) / IF + 1
68:                      INDX_I = IFLG1 - (INDX_J - 1) * IF
69:                      NC=1
70:
71:                      WRITE(*,'(A24,3I6)')'Over flow Z at (K,I,J) :',
73:                      WRITE(*,*)'Within Region :', NREG
74:                      WRITE(*,*)'Computation is unstable.'
75:                    END IF
```

図 4.3-3　WRITE 文括り出し後のコード

(3)　性能分析

図 4.3-4 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	PROC.NAME
40	6.863(3.1)	171.564	558.3	195.4	**0.00**	0.0	0.000	0.000	2.239	0.000	0.000	【チューニング前】
40	0.079(0.0)	1.966	63568.9	18871.6	**99.54**	238.9	0.078	0.000	0.000	0.004	0.012	【チューニング後】

図 4.3-4　WRITE 文括り出し前後 FTRACE 情報

DO ループ全体がベクトル化されたことにより，ベクトル化率が 0.0%から 99.5%に向上し，実行時間は 6.86 秒から 0.08 秒に短縮することができた.

4.3.2 部分ベクトル化の回避

(1) チューニング方針

図 4.3-5にチューニング前のコードを示す. 39行目の変数vmaxは前の繰り返しで定義された値を参照しているためベクトル化の阻害要因となるが, コンパイラは最大値を求める演算をマクロ演算に置き換えてベクトル化を行おうとする. 最大値もしくは最小値を求める演算がある場合, ベクトルパイプライン毎にベクトル演算を行い, 最後に各演算結果のマスクを取るマクロ演算を適用してベクトル化を行う. しかし, 39行目で求めるvmaxの値を40行目で参照する必要があるため, ベクトル化を阻害する依存関係となり, 39行目と40行目をスカラ処理する部分ベクトル化となる. 図 4.3-6に示すチューニング前のFTRACE情報を確認するとベクトル化率が67.5%で十分にベクトル化がされていないことが分かる. ベクトル化を阻害する依存関係を排除し, マクロ演算を適用してループ全体をベクトル化する.

```
!チューニング前
 23:  +-----> do iy = 1,ny
 24:  |V----->   do ix = 1,nx
  :
 37:  ||          absv =max(absvxi,absvxj,absvyi,absvyj)
 38:  ||          cf   = sqrt(cs2(ix,iy)+ca2(ix,iy))
 39:  ||     S    vmax = max(vmax,absv+cf)
 40:  ||     S    dtmin = min(dtmin,dlmin/vmax)
 41:  |V-----   enddo
 42:  +-----  enddo
```

図 4.3-5 部分ベクトル化回避前のコード

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	PROC.NAME
821	26.020(7.2)	31.693	1664.6	496.5	**67.53**	255.9	2.017	0.001	1.503	0.003	1.113	【チューニング前】

図 4.3-6 部分ベクトル化回避前のFTRACE情報

(2) チューニング内容

チューニング前ではvmaxの最大値判定と更新, dtminの最小値判定と更新をループ回数分行っている. dtminの判定式は不変値dlminをvmaxで除算しているため, dtminはvmaxが更新されたときのみdtminが更新される. そのため, DOループでvmaxの最大値を求め, DOループ終了後に一回dtminの判定を行うことで同じ結果が得られる. 図 4.3-7にチューニング後のコードを示す. 前述したとおり, MAX関数で求めるvmaxをMIN関数で参照することが依存関係であるため, MIN関数の演算をDOループの外に移動することで依存関係が解消され, vmaxを求める処理はマクロ演算が適用されてベクトル化が行われる.

```
!チューニング後
23:  +-----> do i y = 1, ny
24:  |V----->   do i x = 1, nx
 :
37:  ||   A      absv = max( absvxi , absvxj , absvyi , absvyj )
38:  ||          cf   = sqrt( cs2( i x, i y) +ca2( i x, i y) )
39:  ||   A      vmax = max( vmax, absv+cf) )
40:  |V-----   enddo
41:  +------ enddo
42:           dtmin = min( dtmin, dl min/vmax)
```

図 4.3-7 部分ベクトル化回避後のコード

(3) 性能分析

図 4.3-8 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	PROC.NAME
821	26.020	(7.2)	31.693	1664.6	496.5	**67.53**	255.9	2.017	0.001	1.503	0.003	1.113	【チューニング前】
821	2.273	(0.8)	2.769	14592.6	4215.6	**99.19**	255.9	1.971	0.000	0.124	0.019	1.029	【チューニング後】

図 4.3-8 部分ベクトル化回避前後の FTRACE 情報

部分ベクトルが解消しベクトル化率が 67.5%から 99.2%に改善したことで実行時間は 26.0 秒から 2.3 秒に短縮することができた.

4.4. 平均ベクトル長の拡大の事例

4.4.1. ループに関する情報の追加による最適化の促進

(1) チューニング方針

多重ループの最内にあるループが自動ベクトル化の対象となるため, 最内ループのベクトル長が短い場合, 十分な性能を発揮できない. 図 4.4-1 のチューニング前のコードではループの中に配列式や配列関数を使用しており, その部分が最内ループとなりベクトル化の対象となる. 図 4.4-2 の FTRACE 情報では平均ベクトル長が 67.0 であり十分なベクトル長でないことが分かる. ここでは, コンパイラがループ長を判断してベクトル長を伸ばす方法で最適化を行う. ループの演算処理量が多いなどの問題からこのような最適化が行われない場合もあるが, ループに関する情報をコンパイラに明示することでベクトル長を伸ばす最適化が行われる場合がある. このような高速化の事例を以下に示す.

```
!チューニング前
  5:            real*8:: y(Nch), xp(Nbim,Nch,Np)
  6:            real*8:: logLhTP(Np), dy(Nch)
  :
 15: V------>   do ip=1,Np
 16: |V=====A    dy = y - xp(1,:,ip)  &
 17: |                  - xp(Nd+1,:,ip) - xp(Nd+Ns,:,ip)
 18: |
 19: |V=====A    logLhTP(ip) = - 0.5d0 * ( &
 20: |                    dble(Nch) * log( 2.0d0 * pi )          &
 21: |                  + log( product( xp(Nbim,:,ip) ))         &
 22: |                  + sum( dy * dy / xp(Nbim,:,ip) )         &
 23: |                    )
 24: V------    enddo
```

図 4.4-1 ループに関する情報を追加する前のコード

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

      60   73.899( 73.1)  1231.647  2404.9  576.8 89.76  67.0   63.853  0.000  0.000   0.001  41.961 【チューニング前】
```

図 4.4-2 ループに関する情報を追加する前の FTRACE 情報

(2) チューニング内容

ベクトル化対象となる配列式と配列関数のループ長(変数 Nch)は 3, 外側 ip の DO ループの長さは 1,280,000 であり, この情報を追加することでコンパイラはより効率の良い最適化を実施する. 図 4.4-3にチューニング後コードを示す. PARAMETER文を追加することでコンパイラは最内DOループのループ長が 3 であることを認識するため, ループの展開を行い, 外側のループでベクトル化を行う. ただし, このチューニング内容は変数 Nch, Np の値が不変であることを前提としている場合のみ使用することが出来るので注意が必要となる.

```
!チューニング後
  6:            parameter(Nch=3,Np=1280000)
  7:            real*8:: y(Nch), xp(Nbim,Nch,Np)
  8:            real*8:: logLhTP(Np), dy(Nch)

 15: V------> do ip=1,Np
 16: |*=====    dy = y - xp(1,:,ip)
 17: |               - xp(Nd+1,:,ip) - xp(Nd+Ns,:,ip)
 18: |
 19: |*=====    logLhTP(ip) = - 0.5d0 * ( &
 20: |                    dble(Nch) * log( 2.0d0 * pi )          &
 21: |                  + log( product( xp(Nbim,:,ip) ))         &
 22: |                  + sum( dy * dy / xp(Nbim,:,ip) )         &
 23: |                    )
 24: V------  enddo
```

図 4.4-3 ループに関する情報を追加後のコード

(3) 性能分析

図 4.4-4 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 AVER RATIO	V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	PROC.NAME
60	**73.899**(73.1)	1231.647	2404.9	576.8	89.76	**67.0**	63.853	0.000	0.000	0.001	41.961	【チューニング前】
60	**0.289**(1.0)	4.813	23763.3	12232.7	99.60	**256.0**	0.267	0.000	0.000	0.017	0.111	【チューニング後】

図 4.4-4 ループに関する情報を追加する前後の FTRACE 情報

外側のループをベクトル化対象とすることで平均ベクトル長が 67.0 から 256.0 となり, 十分な性能を発揮することで, 実行時間が 73.9 秒から 0.3 秒に高速化された.

4.4.2 部分配列を DO ループに書き替え, ループ交換を実施

(1) チューニング方針

チューニング前コードおよびチューニング前の FTRACE 情報は前項と同じ図 4.4-1 と図 4.4-2 である. 前項では, 変数 Nch, Np の値が不変であるという条件のもと PARAMETER 文を追加することでコンパイラの最適化を促進させた. ここでは変数 Nch, Np の値が不定の場合のチューニング方法を示す.

(2) チューニング内容

変数 Nch, Np の値が不定の場合のチューニング方法を示すが, ここで Nch の値が小さく, Np の値が十分に大きいことが分かっているものとする. 図 4.4-5 にチューニング後コードを示す. 配列式と配列関数を DO ループの形に変形し, ループ分割を行っている. このとき作業配列を新たに用意し使用する. また, Nch＜Np であるため, Np の DO ループがベクトル化の対象となるように最内にループの入れ換えを行っている.

```
!チューニング後
  5:            real*8:: y(Nch), xp(Nblm,Nch,Np)
  6:            real*8:: logLhTP(Np), dy(Nch)
  8:            real*8:: wrk_dy,wrk
  9:            real*8:: wrk_sum(Np),wrk_prd(Np)

 24: V-----> wrk_prd=1.d0
 25: V-----  wrk_sum=0.d0
 26: +-----> do j=1,Nch
 27: |V----> do ip=1,Np
 28: ||      A   wrk_dy=y(j)-xp(1,j,ip)                    &
 29: ||                  -xp(Nd+1,j,ip)-xp(Nd+Ns,j,ip)
 30: ||      A   wrk_prd(ip)=wrk_prd(ip)                   &
 31: ||                    * xp(Nblm,j,ip)
 32: ||      A   wrk_sum(ip)=wrk_sum(ip)                   &
 33: ||                  +(wrk_dy**2/xp(Nblm,j,ip))
 34: |V----   end do
 35: +-----   end do
 36:          wrk=dble(Nch)*log(2.0d0*pi)
 37: V-----> do ip=1,Np
 38: |       A   logLhTP(ip)=-0.5d0*(wrk                  &
 39: |               +log(wrk_prd(ip))+wrk_sum(ip))
 40: V-----   enddo
```

図 4.4-5 部分配列を書き換え後のコード

(3) 性能分析

図 4.4-6 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	PROC.NAME
60	**73.899**(73.1)	1231.647	2404.9	576.8	89.76	**67.0**	63.853	0.000	0.000	0.001	41.961	【チューニング前】
60	**0.314**(1.1)	5.232	25804.5	11742.5	99.54	**256.0**	0.287	0.000	0.000	0.026	0.101	【チューニング後】

図 4.4-6 部分配列を書き換え前後の FTRACE 情報

外側のループをベクトル化対象とすることで平均ベクトル長が 67.0 から 256.0 となり, 実行時間が 73.9 秒から 0.3 秒に短縮された.

4.5. メモリ競合の回避の事例

4.5.1. ADB によるメモリアクセス低減

(1) チューニング方針

4.2.4.(3)で述べているように, ADB を利用することにより, より効率的なメモリアクセスを行うことができる. その事例として, 図 4.5-1 を挙げる.

図 4.5-1 において最内側の DO ループがベクトル化されているが, 配列 b はこのベクトル化されたループに入る度にメモリからのロードを行うため, メモリ競合が発生する. 図 4.5-2 の FTRACE 情報が示すように, チューニング前のコードでは, 実行時間 26.6 秒に対してバンクコンフリクト時間が 18.1 秒を占めている. この配列 b に着目してみると, ベクトル化されている DO ループの外側の DO ループ (DO 変数 l のループ) による繰り返しによって配列の参照位置が変わることはなく, 最内 DO ループにおいては, 同じ添え字 j のデータを毎回メモリからロードすることが分かる. 従って, 配列 b は最内 DO ループにおいて再利用性があることが分かる.

通常, コンパイラはループに対する最適化処理の中で配列の再利用性解析を行う. しかし, 本事例のように, ループからの飛び出しがあるような場合にはコンパイラはループの最適化処理を行わない. そのため, 配列の再利用性解析も行われず, 再利用性がある配列に対してもコンパイラは自動で ADB に載せるための処理を行わない. このような場合, コンパイラ指示行により明示的に配列を ADB に載せることを指示し, ADB 経由でのメモリアクセスを行うことにより, メモリに直接アクセスする頻度を削減し, バンクコンフリクト時間を短縮する.

```
!チューニング前
335:  ||+--->              do l=1,lmax
336:  |||\V-->                do i=is,imax(j)+1
337:  ||||                       if(a(l).lt.b(i,j)) exit
338:  |||\V--                 end do
339:  |||                    is    =i
340:  |||                    iwk(l)=i
341:  ||+----              end do
```

図 4.5-1 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 CONFLICT CPU PORT	BANK CONFLICT NETWORK	PROC.NAME
126	26.574(100.0)	210.906	1166.8	342.1	88.72	114.4	22.488	0.000	0.000	0.012	**18.072**	【チューニング前】

図 4.5-2 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

ON_ADB 指示行を用いて, 配列 b を ADB に載せることを指示する. チューニング後のコードを図 4.5-3 に示す. これにより1 回目のロード命令はメモリから行われるが, 2 回目以降のロード命令は ADB から行われることになる. この結果, メモリへのアクセスを削減し, バンクコンフリクトによる実行時間の増加を解消する.

```
!チューニング後
335:  ||+--->              do l=1,lmax
336:  |||          !cdir on_adb(b)
337:  |||V-->                do i=is,imax(j)+1
338:  ||||    A                 if(a(l).lt.b(i,j)) exit
339:  |||V--                   end do
340:  |||                      is    =i
341:  |||                      iwk(l)=i
342:  ||+---                 end do
```

図 4.5-3 ADB によるメモリアクセス低減後のコード

(3) 性能分析

図 4.5-4 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 CPU PORT	CONFLICT NETWORK	PROC.NAME
126	26.574(100.0)	210.906	1166.8	342.1	88.72	114.4	22.488	0.000	0.000	0.012	**18.072**	【チューニング前】
126	9.547(99.9)	75.771	3247.7	952.3	88.72	114.4	5.455	0.000	0.000	0.012	**1.036**	【チューニング後】

図 4.5-4 ADB によるメモリアクセス低減前後の FTRACE 情報

メモリへのアクセスが減少することにより, バンクコンフリクト時間が短縮され, 実行時間が 26.6 秒から9.5 秒に短縮することができた.

4.5.2 ループ融合によるベクトルロード・ストアの削減

(1) チューニング方針

通常, コンパイラによる最適化では, 同じループ構造の DO ループが連続していると自動でループ融合を行う. ループ融合を行うことで同一の配列データに対するメモリアクセス(ロード, ストア)回数を減らすことができる. 図 4.5-5 にチューニング前のコードを示す. チューニング前コードでは同じループ構造の DO ループが連続しているが OpenMP 指示行を含んでいるためループ融合が行われない. コンパイラは PARALLEL リージョン単位で最適化を行うため, ループ融合が行われず連続する複数の DO ループで同じ配列データのメモリアクセスが発生し, 配列 sm3dh はメモリロード2 回, メモリストア 2 回の命令が発生している. 図 4.5-6 の FTRACE 情報を確認すると実行時間 13.1 秒に対してバンクコンフリクト時間が 7.4 秒を占めているため, ループ融合によりメモリアクセス回数を減らしバンクコンフリクト時間が短縮される.

```
!チューニング前
7414:                !$OMP PARALLEL SHARED (sm3dh,sn3dh,sl3dh,sr3dh,se3dh
  :
7419:                !$OMP DO
7420: P----->              do iz=1,lz
7421: |+----->              do iy=1,ly
7422: ||V---->              do ix=3,lx-2
  :
7437: |||      A               sm3dh(ix,iy,iz)=sm3d(ix,iy,iz)
7438: |||                   &             +sdltxinv*( du(ix-1,iy,iz)-du(ix,iy,iz) )
  :
7443: ||V----               enddo
7444: |+-----              enddo
7445: P------              enddo
7446:                !$OMP END DO NOWAIT
7447:                !$OMP END PARALLEL
  :
7454:                      sdltthalfgx=0.5d0*adltt*sgravx
  :
7457:                !$OMP PARALLEL SHARED (sm3dh,sn3dh,sl3dh,sr3dh,se3dh
  :
7462:                !$OMP DO
7463: P----->              do iz=1,lz
7464: |+----->              do iy=1,ly
7465: ||V---->              do ix=3,lx-2
7466: |||      A               sm3dh(ix,iy,iz)=sm3dh(ix,iy,iz)
7467: |||                   &                 +sdltthalfgx*sr3d(ix,iy,iz)
  :
7473: ||V----               enddo
7474: |+-----              enddo
7475: P------              enddo
7476:                !$OMP END DO NOWAIT
7477:                !$OMP END PARALLEL
  :
7498:                !$OMP PARALLEL SHARED (sm3dh,sn3dh,sl3dh,sr3dh,se3dh
  :
7502:                !$OMP DO
7503: P----->              do iz=1,lz
7504: |+----->              do iy=1,ly
7505: ||V---->              do ix=3,lx-2
7506: |||      A               sr3dhinv=1.0d0/sr3dh(ix,iy,iz)
7510: |||      A               su3dh(ix,iy,iz)=sr3dhinv*sm3dh(ix,iy,iz)
  :
7519: ||V----               enddo
7520: |+-----              enddo
7521: P------              enddo
7522:                !$OMP END DO NOWAIT
7523:                !$OMP END PARALLEL
```

図 4.5-5 ループ融合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 CONFLICT CPU PORT	BANK CONFLICT NETWORK	PROC.NAME
200	**13.111** (2.6)	65.557	36092.3	16377.9	99.26	129.0	13.102	0.002	0.005	**0.802**	**6.589**	【チューニング前】

図 4.5-6 ループ融合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

コンパイラによる自動ループ融合が行われない場合，ユーザチューニングによるループ融合が効果的である．チューニング後コードを図 4.5-7 に示す．連続する複数の DO ループは同じループ構造であり，一つの DO ループに修正することでコンパイラが最適化を行う．ループ融合後の配

列 sm3dh に注目すると，演算で求めた結果をレジスタに格納しその後の演算ではレジスタ内のデータを参照することになるため，メモリアクセス回数はメモリストアの 1 回になり，メモリアクセス回数を削減することができる．表 4.5-1 にチューニング前後のメモリアクセス回数の一覧を示す．

```
!チューニング後
7858:                    sdlt thalfgy=0.5d0*adlt t*sgravy
7859:              !$OMP PARALLEL PRIVATE (ix,iy,iz,sr3dhinv)
7860:              !$OMP DO
7861:  P------>           do iz=1,lz
7862:  |+----->           do iy=1,ly
7863:  ||V---->           do ix=3,lx-2
   :
7868:  |||      A            sm3dh(ix,iy,iz)=sm3d(ix,iy,iz)
7869:  |||               &              +sdlt xinv*( du(ix-1,iy,iz)-du(ix,iy,iz) )
   :
7875:  |||                  sm3dh(ix,iy,iz)=sm3dh(ix,iy,iz)
7876:  |||               &                 +sdlt thalfgx*sr3d(ix,iy,iz)
   :
7881:  |||                  sr3dhinv=1.0d0/sr3dh(ix,iy,iz)
7882:  |||                  su3dh(ix,iy,iz)=sr3dhinv*sm3dh(ix,iy,iz)
   :
7891:  ||V----           enddo
7892:  |+-----           enddo
7893:  P------           enddo
7894:              !$OMP END DO NOWAIT
7895:              !$OMP END PARALLEL
```

図 4.5-7 ループ融合によるメモリアクセス低減後のコード

表 4.5-1 ループ融合によるメモリアクセス低減前後のメモリアクセス命令数

	チューニング前	チューニング後
メモリロード	36	16
メモリストア	17	9

(3) 性能分析

図 4.5-8 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  PROC.NAME
           TIME[sec](  %)   [msec]                        RATIO V.LEN     TIME   MISS    MISS CPU PORT NETWORK

      200  13.111( 2.6)     65.557  36092.3  16377.9 99.26 129.0    13.102  0.002   0.005   0.802  6.589 【チューニング前】
      200   8.791( 1.8)     43.953  52951.2  24427.6 99.47 129.0     8.770  0.002   0.003   0.769  3.746 【チューニング後】
```

図 4.5-8 ループ融合によるメモリアクセス低減前後の FTRACE 情報

メモリアクセス回数が減少することでバンクコンフリクト時間が削減され，実行時間が 13.1 秒から 8.8 秒に短縮することができた．

4.5.3. 内側ループの展開によるメモリアクセスの削減

(1) チューニング方針

図 4.5-9 にチューニング前のコードを示す．最内側 n の DO ループの長さ(変数 nmax)は PARAMETER 文で 5 と宣言されており，ベクトル化の対象ループとなるが 121,122 行目の変数 vcx1, vcy2 はベクトル化の阻害要因となるため，コンパイラは最内側 n の DO ループでベクトル化を行わない．一つ外側 ic の DO ループの長さは 4 のため，コンパイラは更に外側 i の DO ループ

でベクトル化を行う. iの DO ループのベクトル化に伴い, ループ分割とループ入れ換えを行う. ループ分割では, iの DO ループのみに依存する演算とi, ic, nの DO ループに依存する演算に分割し, 演算結果を作業配列に格納する. 次にi, ic, nの DO ループの演算では, iの DO ループを内側に移動し事前に求めた作業配列の値を使用してベクトル処理する. このように外側の DO ループをベクトル化することで作業配列を用いたループ分割とループ入れ換えを行うためメモリアクセスの回数が増大する. 図 4.5-10 の FTRACE 情報では, 実行時間 267 秒の内バンクコンフリク ト時間が 196 秒を占めるため, メモリアクセス回数を減らしてバンクコンフリク ト時間を短縮することを検討する.

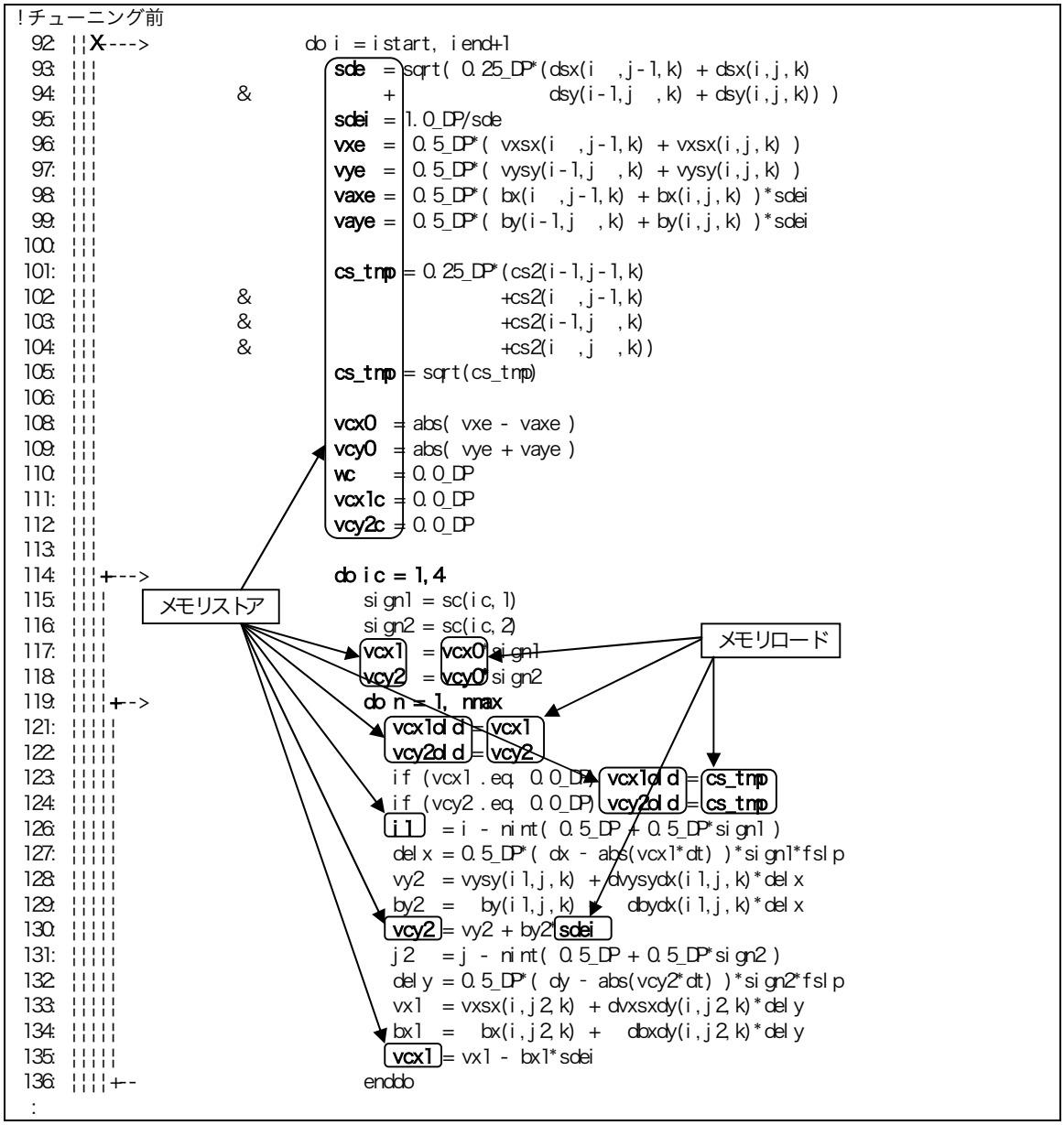


図 4.5-9 内側ループの展開によるメモリアクセス低減前のコード

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

       50  266.912( 17.9)  5338.239  22400.9  6851.1 99.59 213.8   265.674  0.002   0.002   17.916  178.408 【チューニング前】
```

図 4.5-10 内側ループの展開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

内側の DO ループの長さがコンパイル時に判明しているため, EXPAND, UNROLL 指示行を挿入しループの展開を行う. 内側ループを展開することで単純なベクトルループとなるため, ループ分割とループ入れ換えのための作業配列を使用する必要が無く, レジスタのデータで演算を行うことが可能になる. 作業配列を使用しないためメモリアクセスの回数を削減することができる. 図 4.5-11 にチューニング後コード, 表 4.5-2 にチューニング前後のメモリアクセス回数の一覧を示す.

```
!チューニング後
 93: ||V--->              do i = istart, iend+1
 94: |||                     sde  = sqrt( 0.25_DP*(dsx(i  ,j-1,k) + dsx(i,j,k)
 95: |||             &            +              dsy(i-1,j  ,k) + dsy(i,j,k)) )
 96: |||                     sdei = 1.0_DP/sde
 97: |||                     vxe  = 0.5_DP*( vxsx(i  ,j-1,k) + vxsx(i,j,k) )
 98: |||                     vye  = 0.5_DP*( vysy(i-1,j  ,k) + vysy(i,j,k) )
 99: |||                     vaxe = 0.5_DP*( bx(i  ,j-1,k) + bx(i,j,k) )*sdei
100: |||                     vaye = 0.5_DP*( by(i-1,j  ,k) + by(i,j,k) )*sdei
101: |||
102: |||                     cs_tmp = 0.25_DP*(cs2(i-1,j-1,k)
103: |||             &                        +cs2(i  ,j-1,k)
104: |||             &                        +cs2(i-1,j  ,k)
105: |||             &                        +cs2(i  ,j  ,k))
106: |||                     cs_tmp = sqrt(cs_tmp)
107: |||
109: |||                     vcx0  = abs( vxe - vaxe )
110: |||                     vcy0  = abs( vye + vaye )
111: |||                     wc    = 0.0_DP
112: |||                     vcx1c = 0.0_DP
113: |||                     vcy2c = 0.0_DP
114: |||
115: |||         !cdir unroll=4
116: |||*--->                do ic = 1,4
117: ||||                      sign1 = sc(ic,1)
118: ||||                      sign2 = sc(ic,2)
119: ||||                      vcx1  = vcx0*sign1
120: ||||                      vcy2  = vcy0*sign2
121: ||||         !cdir expand=nmax
122: ||||*-->                  do n = 1, nmax
124: |||||                       vcx1old = vcx1
125: |||||                       vcy2old = vcy2
126: |||||                       if (vcx1 .eq. 0.0_DP) vcx1old = cs_tmp
127: |||||                       if (vcy2 .eq. 0.0_DP) vcy2old = cs_tmp
129: |||||                       i1  = i - nint( 0.5_DP + 0.5_DP*sign1 )
130: |||||                       delx = 0.5_DP*( dx - abs(vcx1*dt) )*sign1*fslp
131: |||||                       vy2  = vysy(i1,j,k) + dvysydx(i1,j,k)*delx
132: |||||                       by2  =  by(i1,j,k) +  dbydx(i1,j,k)*delx
133: |||||                       vcy2 = vy2 + by2*sdei
134: |||||                       j2  = j - nint( 0.5_DP + 0.5_DP*sign2 )
135: |||||                       dely = 0.5_DP*( dy - abs(vcy2*dt) )*sign2*fslp
136: |||||                       vx1  = vxsx(i,j2,k) + dvxsxdy(i,j2,k)*dely
```

```
137: |||||                          bx1  =   bx(i,j2,k) +   dbxdy(i,j2,k)*dely
138: |||||                          vx1 = vx1 - bx1*sdei
139: ||||*--                      enddo
 :
```

図 4.5-11 内側ループの展開によるメモリアクセス低減後のコード

表 4.5-2 内側ループの展開によるメモリアクセス低減前後のメモリアクセス命令数

	チューニング前	チューニング後
メモリロード	471	332
メモリストア	329	4
間接参照	160	0

(3) 性能分析

図 4.5-12 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  PROC.NAME
           TIME[sec](  %)   [msec]                          RATIO  V.LEN     TIME    MISS     MISS  CPU PORT  NETWORK

      50  266.912( 17.9)  5338.239  22400.9  6851.1  99.59  213.8   265.674   0.002    0.002   17.916   178.408  【チューニング前】
      50   82.086(  8.5)  1641.720  46808.5 22759.7  99.46  214.4    70.789   3.235    0.002    0.086    14.105  【チューニング後】
```

図 4.5-12 内側ループの展開によるメモリアクセス低減前後の FTRACE 情報

メモリアクセス回数が減少することでバンクコンフリクト時間が短縮され, 実行時間が 266.9 秒から 82.1 秒に高速化することができた.

4.5.4. 複素数のバンク競合回避

(1) チューニング方針

図 4.5-13 に FTRACE 情報を示す. FTRACE 情報では, 実行時間に対してバンクコンフリクトの占める割合が大きいことが分かる. 該当のサブルーチンでは複素数型配列を利用しているために飛びアクセスが発生していると考えられる. このため, 複素数を実数部と虚数部に分け実数型配列に保存し, メモリ上の連続する領域へのアクセスとすることにより, 速度低下やバンクコンフリクトを招く要因となりうる飛びアクセスの削減が期待できる.

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

   44000 1453.723( 41.5)    33.039  31198.3 13481.2  99.62  255.9  1451.882   0.380    0.709  255.084  821.097  【チューニング前】
```

図 4.5-13 複素数のバンク競合回避前の FTRACE 情報

```
!チューニング前
   951: V------>              DO 101 JG=1,NXYZ
   952: V-----          101   RHO2(JG)=(0.D0,0.D0)
    :
   955:               *VDIR NODEP(RHO1)
   956: V------>              DO 100 IG=1,NXYZ
   957: |                     JG=J2G(IG)
   958: V-----          100   RHO1(JG)=P(IG)
    :
   991: V------>           do ig=1,nxyz
   992: |                  VG(ig)=VGG(ig)+Vmoc(ig)
   993: V-----             enddo
    :
  1009: V------>              DO 300 I=1,NXYZ
  1010: |                     fac=dt*dreal( vg(i) )
  1011: V-----          300   RHO2(I)=dcmplx( dcos(fac),-dsin(fac) )*RHO1(I)
    :
  1028:               *VDIR NODEP(RHO2)
  1029: V------>              DO 110 IG=1,NXYZ
  1030: |                     JG=J2G(IG)
  1031: V-----          110   P(IG)=RHO2(JG)
```

図 4.5-14 複素数のバンク競合回避前の前コード

(2) チューニング内容

図 4.5-14 にチューニング前の複素数型配列を利用したコード, 図 4.5-15 に最適化後の複素数を実数部と虚数部に分け実数型配列にしたチューニング後コードを示す. チューニング前コードでは倍精度複素数型を使用し配列 RHO1, RHO2 に演算を行っている. この該当箇所では倍精度実数型を使用し実数部は RHO1_R, RHO2_R とし, 虚数部は RHO1_I ,RHO2_I にそれぞれ分け演算を行い処理するようにする.

```
!チューニング後
   953: V------>              DO  JG=1,NXYZ
   954: |                       RHO2_R(JG)=real(0.D0)
   955: |                       RHO2_I(JG)=aimag((0.D0,0.D0))
   956: V-----                 ENDDO
    :
   968:               *CDIR NODEP
   969: V------>              DO  IG=1,NXYZ
   970: |                       JG=J2G(IG)
   971: |                       RHO1_R(JG)=dble(P(IG))
   972: |                       RHO1_I(JG)=aimag(P(IG))
   973: V-----                 ENDDO
    :
   984: V------>           do I=1,nxyz
   985: |                    VG_R(I)=VGG(I)+Vmoc(I)
   986: |                    fac=dt*VG_R(I)
   987: |                    RHO2_R(I)=(dcos(fac)*RHO1_R(I))-((-dsin(fac))*RHO1_I(I))
   988: |                    RHO2_I(I)=(dcos(fac)*RHO1_I(I))-((dsin(fac))*RHO1_R(I))
   989: V-----              enddo
    :
   995:               *CDIR NODEP
   996: V------>              DO  IG=1,NXYZ
   997: |                       JG=J2G(IG)
   998: |                       P(IG)=dcmplx(RHO2_R(JG),RHO2_I(JG))
   999: |                       VG(IG)=dcmplx(VG_R(IG),0.0d0)
  1000: V-----                 ENDDO
```

図 4.5-15 複素数のバンク競合回避後のコード

(3) 性能分析

図 4.5-16 は，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	PROC.NAME
44000	**1453.723**(41.5)	33.039	31198.3	13481.2	99.62	255.9	1451.882	0.380	0.709	255.084	821.097	【チューニング前】
44000	**996.058**(34.9)	22.638	45680.0	19675.5	99.63	255.9	993.704	0.676	1.039	234.856	466.970	【チューニング後】

図 4.5-16 複素数のバンク競合回避前後の FTRACE 情報

メモリ上の連続する領域へのアクセスすることにより，不連続なメモリアクセスが軽減され，実行時間が 1,453 秒から 996 秒に短縮することができた．

4.6. ベクトル演算の効率化の事例

4.6.1. 総和演算の効率化

(1) チューニング方針

ベクトルループ内の総和演算は一回前の繰り返しの結果を参照するためベクトル化の阻害要因となる．しかし，コンパイラは総和演算を特別なパターンであることを認識し，専用のベクトル命令(総和型マクロ演算)を用いることによりベクトル化を行う．図 4.6-1 はチューニング前のコードである．配列 Wn は i の次元を持たないため i の DO ループをベクトル化すると，コンパイラは総和型マクロ演算を用いてベクトル化を行う．この総和型マクロ演算により実行を高速化できるが，総和演算を含む多重ループの場合，ループを入れ替え，総和型マクロ演算をベクトル演算にすることによりさらなる高速化が可能となる場合がある．このような事例を以下に示す．

```
!チューニング前
11:  +------>       do j=1,ny
12:  |V----->       do i=1,nx
13:  |||
14:  |||              Wm(j) = Wm(j) + Wn(i,j)
15:  |||
16:  |V-----         end do
17:  +------         end do
```

図 4.6-1 総和演算の効率化前のコード

(2) チューニング内容

図 4.6-2 にループ入れ換え後のコードを示す．ループ入れ換えにより j の DO ループでベクトル化することで配列 Wm のメモリアクセスがストライドになることに注意されたい．

```
!チューニング後
12:  +------>       do i=1,nx
13:  |V----->       do j=1,ny
14:  |||
15:  |||              Wm(j) = Wm(j) + Wn(i,j)
16:  |||
17:  |V-----         end do
18:  +------         end do
```

図 4.6-2 総和演算の効率化後のコード

(3) 性能分析

図 4.6-3 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  PROC.NAME
         TIME[sec](  %)     [msec]                  RATIO V.LEN    TIME   MISS    MISS CPU PORT NETWORK

        1  232.086( 71.8) 232086.438 17813.0  8620.3 99.30 173.6  232.083  0.001   0.002   0.000  32.327 【チューニング前】
        1   91.148( 28.2)  91148.138 36054.1 20772.5 99.11 191.0   91.147  0.000   0.001  27.536   4.160 【チューニング後】
```

図 4.6-3 総和演算の効率化前後の FTRACE 情報

チューニングを行うことで実効性能が向上し, 実行時間が 232.1 秒から91.1 秒に高速化することができた.

4.6.2 IF 文の括り出しによる演算数の削減

(1) チューニング方針

ベクトルループ内に IF 文がある場合, SX-9 では IF 文の真偽に関わらず全ての演算を行い最後に IF 文の真にあたる結果のみを採用する方法でベクトル処理を行う. 図 4.6-4 にチューニング前のコードを示す.

```
!チューニング前
12  +------>       do j=1,my
13  |V----->       do i=1,mx
14  |||              if(itbl(i)==1)then
15  |||                  Vm(i,j) = Vm(i,j) + Vwk(i,j)
16  |||              endif
17  |V-----         end do
18  +------         end do
```

図 4.6-4 IF 文の括り出しによる演算数削減前のコード

(2) チューニング内容

図 4.6-5 にチューニング後のコードを示す. IF 文の判定式である配列 itbl はベクトル化対象であるi の DO ループの次元しか持たないため, IF 文を含めたループ入れ換えを行うことでベクトルループ内に IF 文の処理が無くなる. ベクトル計算機では, ベクトルループ内に IF 文がある場合, 条件式の真偽に関わらず両方の演算を行い, 最後に条件式が真となる場合の結果を選択する. 両方の演算を行うため演算数は増加するが, ベクトル演算を行うことにより性能が向上する. ループ入れ換えを行うことにより, ベクトル演算の効率を低下させることなく演算数を削減し, 実行時間を短縮することができる場合がある. このような高速化の事例を以下に示す. なお, ループの入れ換えにより, 配列 Wn, Wm のメモリアクセスがストライドとなるため, メモリアクセスの観点から性能が低下する可能性があり, 実際に高速化できるかについては確認が必要である.

```
!チューニング後
12  +------>       do i=1,mx
13  |                if(itbl(i)==1)then
14  |V----->       do j=1,my
15  ||                 Vm(i,j) = Vm(i,j) + Vwk(i,j)
16  |V-----         end do
17  |                endif
18  +------         end do
```

図 4.6-5 IF 文の括り出しによる演算数削減後のコード

(3) 性能分析

図 4.6-6 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 AVER RATIO	V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	PROC.NAME
1	**98.349**(68.2)	98349.331	36691.7	**20319.8**	98.99	191.0	98.348	0.000	0.001	22.750	4.486	【チューニング前】
1	**45.849**(31.8)	45848.683	36026.1	**20756.3**	99.11	191.0	45.847	0.000	0.001	13.683	2.121	【チューニング後】

図 4.6-6 IF 文の括り出しによる演算数削減前後の FTRACE 情報

ベクトル演算量(実行時間×実効性能)が約 1/2 となるため, 実行時間が 98.3 秒から 45.9 秒に短縮することができた.

4.6.3. ライブラリの置き換えによる高速化

(1) チューニング方針

SX-9 では, HPC 用に高度に最適化された数学ライブラリ集の Mathkeisan が利用できる. BLAS/LAPACK も最適化されており, プログラムの高速化が容易に行える.

図 4.6-7 にチューニング前のコードを示す. このコードでは, LAPACK のサブルーチン zheev が呼び出されている. 図 4.6-8 のチューニング前の性能情報を確認すると, ベクトル化率が 98.17%となっている. zheev は QR 法を用いてエルミート行列の固有値・固有ベクトルを求めるが, 分割統治法でこれらを求めるライブラリzheevd に置き換えることでベクトル化率の向上と計算時間の短縮を図る.

なお, 分割統治法は QR 法より高速と知られているが, QR 法の演算量が行列の次数の 3 乗のオーダーであるのに対し, 分割統治法の演算量は 2 乗から 3 乗のオーダーとばらつきがあることから, 場合によっては, 有意な性能差がみられないケースもありうる. また, 必要となるメモリ領域の大きさについては, QR 法は次数の 1 乗のオーダーであるのに対し, 分割統治法は 2 乗のオーダーとなる点も考慮する必要がある.

```
!チューニング前
983:          call zheev('V','U',nb*2,v2,nb*2,eig2,work2,2*(nb*2)-1,rwork2, info)
```

図 4.6-7 ライブラリ置き換え前のコード

FREQUENCY	EXCLUSIVE TIME[sec] (%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP AVER RATIO	V.LEN	VECTOR TIME	I-CACHE MISS	O-CACHE MISS	BANK CONFLICT CPU PORT	BANK CONFLICT NETWORK	PROC.NAME
4	78.111(5.0)	19527.870	7546.6	4231.4	98.17	210.4	27.766	0.022	23.009	3.657	18.205	【チューニング前】

図 4.6-8 ライブラリ置き換え前の FTRACE 情報

(2) チューニング内容

図 4.6-9 にチューニング後のコードを示す. チューニング前のコードで呼び出されていた LAPACK のサブルーチン zheev を zheevd に置き換えている. zheev と zheevd は呼び出しの際の引数が異なり, 作業用の配列も 2 つ増える. 作業配列のサイズは計算前に事前に一度 zheevd を呼び出すことで, 計算に最適なサイズを取得することができる. 取得したサイズの作業配列を確保し, zheevd の呼び出す.

```
!チューニング後
184:                    lwork2=-1
185:                    lrwork2=-1
186:                    liwork2=-1
187:                    call zheevd('V','U',nb*2,v2,nb*2,eig2,work2,lwork2,rwork2,
188:          &              lrwork2,iwork2,liwork2,info)
189:                    lwork2=work2(1)
190:                    lrwork2=rwork2(1)
191:                    liwork2=iwork2(1)
192:                    deallocate (work2,rwork2,iwork2)
193:                    allocate (work2(lwork2),rwork2(lrwork2),iwork2(liwork2))
 :
986:                 call zheevd('V','U',nb*2,v2,nb*2,eig2,work2,lwork2,rwork2,
987:          &              lrwork2,iwork2,liwork2,info)
```

図 4.6-9 ライブラリ置き換え後のコード

(3) 性能分析

図 4.6-10 はチューニング後の性能情報である. 適切なライブラリを選択することでベクトル化率が 98.17%から 99.38%へ向上し, 78.11 秒から 6.954 秒に短縮することができた. また, zheev から zheevd に置換することによる計算結果への影響もみられなかった.

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	PROC.NAME
4	78.111(5.0)	19527.870	7546.6	4231.4	98.17	210.4	27.766	0.022	23.009	3.657	18.205	【チューニング前】
4	6.954(0.5)	1738.465	53869	31038.0	99.38	221.5	6.729	0.014	0.059	1.026	2.812	【チューニング後】

図 4.6-10 ライブラリ置き換え前後の FTRACE 情報

ここではライブラリルーチンの置き換えによる高速化の事例を挙げたが, 置換する際には, 置換によるメリット・デメリットを把握した上で置換を行い, 置換後の計算結果の妥当性も必ず確認する必要がある.

4.6.4. ファイルアクセスの高速化

(1) チューニング方針

図 4.6-11 に FTRACE 情報を示す. FTRACE 情報では, ベクトル化率が 1.2% となっている. 詳細な解析により, 図 4.6-12 に DO ループの中でファイルからデータを読み込んだ直後, 読み込んだデータに対して演算が実行されていることが分かった. そこで, ファイルの読み込みと演算を別々に処理するよう最適化を行う.

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	PROC.NAME
1	219.190(12.5)	219189.905	469.7	4.1	1.20	248.1	0.082	34.416	5.627	0.395	0.057	【チューニング前】

図 4.6-11 ファイルアクセス高速化前の FTRACE 情報

```
!チューニング前
  2776: +------>          DO 450 IK = 1, NUMK
   :    |
  2783: |                 if ( my_rank.eq.0 ) then
  2784: |+----->           do icpu=0,ncpu
  2785: ||                 nbleng=nend(icpu)-nbegin(icpu)+1
  2786: ||+---->           do 451 ib=1,nbleng
  2787: |||                read(22)( rho1(ig),rho2(ig),ig=1,nxyz )
  2788: |||\/--->           do ig=1,nxyz
  2789: ||||                coef0(ig,ib,ik)=dcmplx( rho1(ig),rho2(ig) )
  2790: |||\/--             enddo
  2791: ||+---         451 continue
  2792: ||                 if ( icpu.eq.0 ) then
  2793: ||+---->           do ib=1,nbleng
  2794: |||\/--->           do ig=1,nxyz
  2795: ||||                coef(ig,ib,ik)=coef0(ig,ib,ik)
  2796: |||\/--             enddo
  2797: ||+---             enddo
  2798: ||                 else
  2799: ||                  call MPI_Send(coef0(1,1,ik),nxyz*nbleng,
  2800: ||                & MPI_DOUBLE_COMPLEX,icpu,tag,MPI_COMM_WORLD,ierr)
  2801: ||                 endif
  2802: ||
  2803: |+-----            enddo  ! end of icpu loop
  2804: |                  else
  2805: |                  nbleng=nend(my_rank)-nbegin(my_rank)+1
  2806: |                  call MPI_Recv(coef0(1,1,ik),nxyz*nbleng
  2807: |                &,MPI_DOUBLE_COMPLEX,0,tag,MPI_COMM_WORLD,status,ierr)
  2808: |+---->           do ib=1,nbleng
  2809: ||\/--->           do ig=1,nxyz
  2810: |||                coef(ig,ib,ik)=coef0(ig,ib,ik)
  2811: ||\/---            enddo
  2812: |+-----            enddo
  2813: |                 endif
  2814: +-----        450 CONTINUE     ! ik roop
```

図 4.6-12 ファイルアクセスの高速化前のコード

(2) チューニング内容

図 4.6-12 にチューニング前コード, 図 4.6-13 にチューニング後コードを示す. チューニング前コードでは read(22) が DO ループの中にあり, その中でさらに演算を行っている. この該当箇所に対して read(22) をDO ループの外へ移動し, 演算に関連する部分の配列を1次元配列から2次元配列に変更しファイルの読み込みと演算を別々に処理するようにする.

```
!チューニング後
 2776: +------>          DO 450 IK = 1, NUMK
  :    |
 2783: |                 if ( my_rank.eq.0 ) then
 2784: |+----->            do ib=1,nend(ncpu)
 2785: ||                   read(22)(tmp_22(ig,ib),ig=1,nxyz)
 2786: |+-----             enddo
 2787: |                   icnt=0
 2788: |+----->            do icpu=0,ncpu
 2789: ||                   nbleng=nend(icpu)-nbegin(icpu)+1
 2790: ||+---->             do ib=1,nbleng
 2791: |||\/-->               do ig=1,nxyz
 2792: ||||                    coef0(ig,ib,ik)=tmp_22(ig,ib+icnt)
 2793: |||\/--                 enddo
 2794: ||+---                 enddo
 2795: ||                     icnt=icnt+nbleng
 2796: ||                     if ( icpu.eq.0 ) then
 2797: ||+---->                do ib=1,nbleng
 2798: |||\/-->                  do ig=1,nxyz
 2799: ||||                       coef(ig,ib,ik)=coef0(ig,ib,ik)
 2800: |||\/--                   enddo
 2801: ||+---                   enddo
 2802: ||
 2803: ||                       else
 2804: ||                       call MPI_Send(coef0(1,1,ik),nxyz*nbleng,
 2805: ||                  &      MPI_DOUBLE_COMPLEX,icpu,tag,MPI_COMM_WORLD,ierr)
 2806: ||
 2807: ||                       endif
 2808: |+-----              enddo  ! end of icpu loop
  :    |
 2811: |                   else
 2812: |                   nbleng=nend(my_rank)-nbegin(my_rank)+1
 2813: |                   call MPI_Recv(coef0(1,1,ik),nxyz*nbleng,
 2814: |                  &   MPI_DOUBLE_COMPLEX,0,tag,MPI_COMM_WORLD,status,ierr)
 2815: |+----->            do ib=1,nbleng
 2816: ||\/--->              do ig=1,nxyz
 2817: |||                     coef(ig,ib,ik)=coef0(ig,ib,ik)
 2818: ||\/---               enddo
 2819: |+-----              enddo
 2820: |                   endif
 2821: +-----        450 CONTINUE      ! ik roop
```

図 4.6-13 ファイルアクセスの高速化後のコード

(3) 性能分析

図 4.6-14 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	PROC.NAME
1	219.190(12.5)	219189.905	469.7	4.1	**1.20**	248.1	0.082	34.416	5.627	0.395	0.057	【チューニング前】
1	0.095(0.0)	94.761	13163.1	0.0	**98.99**	248.0	0.087	0.001	0.001	0.017	0.061	【チューニング後】

図 4.6-14 ファイルアクセスの高速化前後の FTRACE 情報

ファイルの読み込みと演算を別々に処理することにより, ベクトル化率が1.2%から98.99%に改善したことで実行時間が219.190秒から0.095秒に短縮することができた.

4.7. 複合問題の事例

4.7.1. 事例 1

(1) チューニング方針

図 4.7-1 にチューニング前のコードを示す. サブルーチン sub の二重 DO ループ内にはユーザ関数 func1, func2 の呼び出しがある. func1 ではベクトル化の阻害要因である GOTO 文による繰り返しを行っているためベクトル化されていない. また, 一要素の乱数値を生成するユーザ関数 ran2 の呼び出しを行っている. func2 ではベクトル化を行っているもののベクトル長が短く十分な性能を発揮できていない. 図 4.7-2 の FTRACE 情報ではベクトル化率 79.5%, 平均ベクトル長 59.0 となっている.

チューニングを行うにあたり, サブルーチン sub, ユーザ関数 func1, func2 のそれぞれのループ長からベクトル化の対象ループを選別する. sub の i の DO ループの長さは 20,062, m の DO ループの長さは 50 であり, func1 と func2 内の繰り返し数は不定のため, i の DO ループでベクトル化を行うことを考える. そのためには手動インライン展開, ループ分割, ループ入れ換えの手法を用いてチューニングを行う. また, 乱数生成方法の変更を行うことで更なる高速化を図る. 実際のチューニング手順を段階に分けて後述する.

```
!チューニング前
 1:          subroutine sub(nn,m0,m1,iseed,dvar,dimp,out1,out2)
 2:            integer,parameter :: nmax=20100
 3:            integer :: nbin,m0,m1,iseed
 4:            double precision :: dvar,out1,out2
 5:            double precision :: dimp(nmax,*)
 6:
 7:            integer :: i,j,m
 8:            double precision :: p,tmp(nmax)
 9:
10: +------>   do i=1,nn
11: |+----->   do m=m1,90
12: ||           j=m-m0
13: ||           p=dvar*dimp(i,j)
14: ||           tmp(i)=func1(p,iseed)
15: ||           out1=out1+tmp(i)
16: ||           out2=out2+tmp(i)*log(p)-p-func2(int(tmp(i)))
17: |+-----    enddo
18: +-----    enddo
19:
20:           return
21:         end subroutine
------
22:         function func1(xm,idum)
23:         INTEGER idum
24:         REAL func1,xm
25:         REAL g,em,t
26:
27:         g=exp(-xm)
28:         em=-1
29:         t=1.
30:       2 em=em+1.
31:         t=t*ran2(idum)
32:         if (t.gt.g) goto 2
33:         func1=em
34:
35:         return
36:         end function
```

```
-----
37:           function func2(n)
38:           integer n
39:           if ( n .eq. 0 ) then
40:             func2=0.0
41:           else
42:             func2=0.0
43:  V------>   do i=1,n
44:  |           func2=func2+log(float(i))
45:  V------    enddo
46:           endif
47:
48:           return
49:           end function
```

図 4.7-1 複合事例 1 のチューニング前コード

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	PROC.NAME
10	42.140(100.0)	4214.041	698.7	71.5	79.56	59.0	4.912	0.001	1.284	0.015	3.685	【チューニング前】

図 4.7-2 複合事例 1 のチューニング前 FTRACE 情報

(2) チューニング内容

<STEP1 ループ入れ換えとループ分割>

サブルーチン sub の二重 DO ループにおいて, ベクトル化対象とする i の DO ループを内側に入れ換えて, 今後のチューニングの簡潔化のために内側 i の DO ループをループ分割する. ループ分割を行う場合, データの依存性に注意が必要だが, func1 で求める配列 tmp は i の DO ループの次元を持っているため, 特別な追加処理は必要なくループ分割が可能となる.

```
!STEP1
 1:           subroutine sub(nn,m0,m1,iseed,dvar,dimp,out1,out2)
 :
10:  +------>   do m=m1,90
11:  |            j=m-m0
12:  |+----->   do i=1,nn
13:  ||           p=dvar*dimp(i,j)
14:  ||           tmp(i)=func1(p,iseed)
15:  |+-----    end do
16:  |+----->   do i=1,nn
17:  ||           p=dvar*dimp(i,j)
18:  ||           out1=out1+tmp(i)
19:  ||           out2=out2+tmp(i)*log(p)-p-func2(int(tmp(i)))
20:  |+-----    enddo
21:  +------    enddo
```

図 4.7-3 [STEP1] ループ入れ換えとループ分割

<STEP2 手動インライン展開>

チューニング方針で示したようにサブルーチン sub の i の DO ループでベクトル化を行うためには, i の DO ループが最内側になるようループ入れ換えを行う必要がある. コンパイラによる自動インライン展開ではループ入れ換えを含む最適化は行われないため, sub で呼び出しているユーザ

関数 func1, func2 を手動でインライン展開を行う. 手動インライン展開を行うことで sub のループ(ループ長 nn の DO ループ)とfunc1 の GOTO 文による繰り返し, func2 のループ(ループ長 n の DO ループ)を入れ換えることが可能になる.

以降, func1 部分のチューニング内容を STEP3, STEP4, STEP5 に func2 部分のチューニングを STEP6, STEP7 に示す.

```
!STEP2
 1:           subroutine sub(nn,m0,m1,iseed,dvar,dimp,out1,out2)
  :
11:  +------>    do m=m1,90
12:  |            j=m-m0
13:  |+----->    do i=1,nn
14:  ||             p=dvar*dimp(i,j)
15:  ||             g=exp(-p)
16:  ||             em=-1
17:  ||             t=1.
18:  ||         2   em=em+1.
19:  ||             t=t*ran2(iseed)
20:  ||             if (t.gt.g) goto 2
21:  ||             tmp(i)=em
22:  |+-----     end do
23:  |V----->    do i=1,nn
24:  ||             p=dvar*dimp(i,j)
25:  ||             out1=out1+tmp(i)
26:  ||       S     n=int(tmp(i))
27:  ||             f=0.0
28:  ||       S     if (n.ne.0) then
29:  ||V--->           do k=1,n
30:  |||                f=f+log(float(k))
31:  ||V---           enddo
32:  ||             endif
33:  ||             out2=out2+tmp(i)*log(p)-p-f
34:  |V-----     enddo
35:  +------     enddo
```

図 4.7-4 [STEP2] 手動インライン展開

<STEP3 ループ分割>

func1 部分の DO ループ内には, 初回に実行される箇所とGOTO 文による繰り返し部分, 結果の格納部分に分けることができる. 処理部分毎に i の DO ループをループ分割する. ループ分割する際はループ内変数から作業配列に変更して値を保持する.

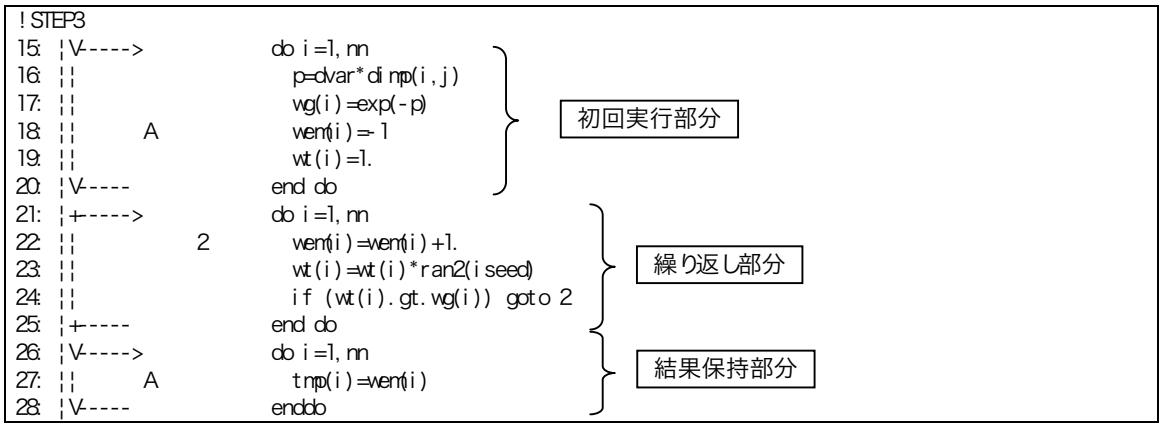


図 4.7-5 [STEP3] DO ループの分割

<STEP4 GOTO 文の書き換え>

GOTO 文による繰り返し部分のチューニング内容を図 4.7-6 に示す. GOTO 文による繰り返しを DO WHILE 文に変更する. その際, 繰り返しの終了条件を判断するため, 作業変数 iflag_all を新たに用意する. i の DO ループ毎に繰り返しの有無を判断するために作業配列 iflag を用意する. チューニング前とチューニング後の処理の流れは以下のとおりとなる.

[チューニング前]

① 演算処理を行う

② IF 文が真であれば①に戻る

[チューニング後]

① 変数 iflag_all, 配列 iflag に初期値 0 を代入する

② 変数 iflag_all の値が 1 以外のとき, 繰り返し処理を続行する

③ 配列 iflag の値が 1 以外のとき, 演算処理を行う

④ IF 文が真であれば iflag に 1 を代入する

⑤ 配列 iflag の値をチェックし, 全ての値が 1 ではなければ②に戻る

```
!STEP4
29: ¦  iflag_all=0
30: ¦V═══  iflag=0
31: ¦+----->  do while(iflag_all.ne.1)
32: ¦¦+--->  do i=1,nn
33: ¦¦¦  if(iflag(i).ne.1)then
34: ¦¦¦  wem(i)=wem(i)+1.
35: ¦¦¦  wt(i)=wt(i)*ran2(iseed)
36: ¦¦¦  if (wt(i).le.wg(i)) iflag(i)=1
37: ¦¦¦  end if
38: ¦¦+---  end do
39: ¦¦V--->  do i=1,nn
40: ¦¦¦  if(iflag(i).ne.1) goto 20
41: ¦¦V---  end do
42: ¦¦  iflag_all=1
43: ¦¦  20  continue
44: ¦+----  end do
```

図 4.7-6 [STEP4] DO WHILE 文への置き換え

<STEP5 乱数生成方法の変更>

DO ループ内にユーザ関数 ran2(乱数値の生成)の呼び出しがあるため，ベクトル化が行われないが事前に乱数値を生成することでベクトル化が可能になる.

```
!STEP5
29:  ¦                    iflag_all=0
30:  ¦V=====              iflag=0
31:  ¦+----->             do while(iflag_all.ne.1)
32:  ¦¦                     if(icnt+nn.ge.inum) call make_random_number
33:  ¦¦V--->                do i=1,nn
34:  ¦¦¦                      if(iflag(i).ne.1)then
35:  ¦¦¦     A                  wem(i)=wem(i)+1.
36:  ¦¦¦                        icnt=icnt+1
37:  ¦¦¦     A                  wt(i)=wt(i)*wran(icnt) !ran2(iseed)
38:  ¦¦¦     A                  if (wt(i).le.wg(i)) iflag(i)=1
39:  ¦¦¦                      end if
40:  ¦¦V---                 end do
41:  ¦¦V--->                do i=1,nn
42:  ¦¦¦                      if(iflag(i).ne.1) goto 20
43:  ¦¦V---                 end do
44:  ¦¦                     iflag_all=1
45:  ¦¦           20        continue
46:  ¦+-----               end do
```

図 4.7-7 [STEP5] 乱数生成方法の変更

<STEP6 ループ分割>

func2 部分は DO ループによる変数 f の総和演算を行い，変数 f を用いて変数 out1, out2 の結果を求める総和演算処理にわかれているため，処理内容毎に DO ループを分割する. 変数 f は作業配列 wf に変更して値を保持するように変更する.

```
!STEP6
46:  ¦V----->             do i=1,nn
47:  ¦¦                     wf(i) =0.0
48:  ¦V-----              end do
49:  ¦V----->             do i=1,nn
50:  ¦¦       S             n=int(tmp(i))
51:  ¦¦       S             if (n.ne.0) then
52:  ¦¦V--->                  do k=1,n
53:  ¦¦¦                       wf(i)=wf(i)+log(float(k))
54:  ¦¦V---                   enddo
55:  ¦¦                      endif
56:  ¦V-----              enddo
57:  ¦V----->             do i=1,nn
58:  ¦¦       A             p=dvar*dimp(i,j)
59:  ¦¦       A             out1=out1+tmp(i)
60:  ¦¦                     out2=out2+tmp(i)*log(p)-p-wf(i)
61:  ¦V-----              enddo
```

図 4.7-8 [STEP6] DO ループの分割

<STEP7 ループ入れ換え>

配列 wf を求める総和演算部分では，最内側 n の DO ループの長さ(int(tmp(i)))が不明なため，

事前に最大数を計算し, DO ループの入れ換えを行う. ループ入れ換え後には本来のループ回数分の処理を行うために IF 文を挿入して制御する.

```
!STEP7
46:  |                    imax=0
47:  |V---->              do i=1,nn
48:  ||      A               wf(i)=0.0
49:  ||      A               imax=max(imax,int(tmp(i)))
50:  |V-----               end do
51:  |+----->              do k=1,imax
52:  ||V---->                 do i=1,nn
53:  |||     A                 if(int(tmp(i)).ge.k)then
54:  |||     A                   wf(i)=wf(i)+log(float(k))
55:  |||                       endif
56:  ||V-----                 end do
57:  |+------               end do
```

図 4.7-9 [STEP7] DO ループの入れ換え

<最終版>

上述 STEP1 からSTEP7 までを適用した最終ソースコードを図 4.7-10 に示す.

```
!チューニング後
1:                    subroutine sub(nn,m0,m1,iseed,dvar,dimp,out1,out2)
 :
16:  +------>           do m=m1,90
17:  |                    j=m-m0
18:  |V---->             do i=1,nn
19:  ||                   p=dvar*dimp(i,j)
20:  ||      A            wg(i)=exp(-p)
21:  ||      A            wem(i)=-1
22:  ||      A            wt(i)=1.
23:  ||                   end do
24:  ||                   do i=1,nn
25:  ||                   iflag(i)=0
26:  |V-----              end do
27:  |                    iflag_all=0
28:  |+----->             do while(iflag_all.ne.1)
29:  ||                   if(icnt+nn.ge.inum) call make_random_number
30:  ||V---->              do i=1,nn
31:  |||     A              if(iflag(i).ne.1)then
32:  |||     A                wem(i)=wem(i)+1.
33:  |||                      icnt=icnt+1
34:  |||     A                wt(i)=wt(i)*wran(icnt) !ran2(iseed)
35:  |||     A                if (wt(i).le.wg(i)) iflag(i)=1
36:  |||                    end if
37:  ||V----               end do
38:  ||V---->              do i=1,nn
39:  |||     A              if(iflag(i).ne.1) goto 20
40:  ||V----               end do
41:  ||                   iflag_all=1
42:  ||           20      continue
43:  |+-----              end do
44:  |V----->             do i=1,nn
45:  ||      A            tmp(i)=wem(i)
46:  ||                   enddo
47:  ||                   imax=0
48:  ||                   do i=1,nn
49:  ||                   wf(i)=0.0
50:  ||                   imax=max(imax,int(tmp(i)))
51:  |V-----              end do
52:  |+----->             do k=1,imax
53:  ||V---->              do i=1,nn
54:  |||     A              if(int(tmp(i)).ge.k)then
55:  |||     A                wf(i)=wf(i)+log(float(k))
56:  |||                    end if
57:  ||V----               end do
58:  |+-----              end do
59:  |V----->             do i=1,nn
60:  ||      A            p=dvar*dimp(i,j)
61:  ||      A            out1=out1+tmp(i)
62:  ||      A            out2=out2+tmp(i)*log(p)-p-wf(i)
63:  |V-----              enddo
64:  +------             enddo
```

func1 部分

func2 部分

図 4.7-10 複合事例 1のチューニング後コード

(3) 性能分析

図 4.7-11 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	PROC.NAME
10	**42.140**(100.0)	4214.041	698.7	71.5	**79.56**	**59.0**	4.912	0.001	1.284	0.015	3.685	【チューニング前】
10	**0.193**(97.1)	19.277	13250.0	4307.2	**99.12**	**239.3**	0.182	0.000	0.000	0.001	0.120	【チューニング後】

図 4.7-11 複合事例 1 のチューニング前後 FTRACE 情報

ベクトル化の阻害要因を取り除くことでベクトル化率は 79.6%から 99.1%に改善し, ベクトル長の長い DO ループでベクトル化を行うことで平均ベクトル長は 59.0 から 239.3 に改善している. その結果, 実行時間は 42.1 秒から 0.2 秒に短縮することができた.

4.7.2 事例 2

(1) チューニング方針

図 4.7-12 にチューニング前のコードを示す. 配列 t は配列 ip1～ip11, iml～iml1 の値により参照するアドレスが変わる間接参照となる. 図 4.7-13 の FTRACE 情報では実行時間 73 秒に対してバンクコンフリクト時間が 45 秒となっている. 間接参照を取り除きバンクコンフリクト時間を短縮し高速化する手順を説明する.

```
!チューニング前
 112:  +------>          do k=nz_s,nz_e
 113:  |+----->          do j=1,ny
 114:  ||V---->          do i=1,nx
  :
 136:  |||      A        diff_x=-(       +coef2_x11*t(ip11(i),j  ,k  )
 137:  |||               &              +coef2_x10*t(ip10(i),j  ,k  )
  :                                                                    ← 配列 ip1～ip11 の値を参照
 145:  |||               &              +coef2_x2 *t(ip 2(i),j  ,k  )
 146:  |||               &              +coef2_x1 *t(ip 1(i),j  ,k  )
 147:  |||               &         +2.0d0*coef2_x0 *t(i     ,j  ,k  )
 148:  |||               &              +coef2_x1 *t(im1(i),j  ,k  )
 149:  |||               &              +coef2_x2 *t(im2(i),j  ,k  )
  :                                                                    ← 配列 im1～im11 の値を参照
 157:  |||               &              +coef2_x10*t(im10(i),j  ,k  )
 158:  |||               &              +coef2_x11*t(im11(i),j  ,k  ))*alph0
 160:  |||      A        diff_y=-(       coef2_y11*t(i  ,jp11(j),k  )
 161:  |||               &              +coef2_y10*t(i  ,jp10(j),k  )
  :
 169:  |||               &              +coef2_y2 *t(i  ,jp 2(j),k  )
 170:  |||               &              +coef2_y1 *t(i  ,jp 1(j),k  )
 171:  |||               &         +2.0d0*coef2_y0 *t(i  ,j     ,k  )
 172:  |||               &              +coef2_y1 *t(i  ,jm1(j),k  )
 173:  |||               &              +coef2_y2 *t(i  ,jm2(j),k  )
  :
 181:  |||               &              +coef2_y10*t(i  ,jm10(j),k  )
 182:  |||               &              +coef2_y11*t(i  ,jm11(j),k  ))*alph0
 183:  |||
  :
 193:  ++V----           enddo ; enddo ; enddo
```

図 4.7-12 複合事例 2 のチューニング前コード

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

      17   73.223( 28.1)  4307.257 11676.6  5186.6 99.83 240.0   52.196 0.000  20.436   3.408  41.798 【チューニング前】
```

図 4.7-13 複合事例２のチューニング前 FTRACE 情報

(2) チューニング内容

配列 ip1～ip11, iml～iml1 の値は図 4.7-14 の DO ループで設定されている.

```
!配列ip1～ip11, iml～iml1の設定
2711: V------>          do i=1,nx
2712: |                 iml1(i) =i-11
2713: |                 iml0(i) =i-10
 :
2721: |                 im2(i) =i-2
2722: |                 iml(i) =i-1
2723: |                 ip1(i) =i+1
2724: |                 ip2(i) =i+2
 :
2732: |                 ip10(i) =i+10
2733: |                 ip11(i) =i+11
2735: |                 if(ip1(i).gt.nx)ip1(i) =ip1(i) -nx
2736: |                 if(ip2(i).gt.nx)ip2(i) =ip2(i) -nx
 :                                                          } 参照するアドレスが nx を超える場合, 1 に戻る
2744: |                 if(ip10(i).gt.nx)ip10(i) =ip10(i) -nx
2745: |                 if(ip11(i).gt.nx)ip11(i) =ip11(i) -nx
2746: |                 if(iml(i).lt.1 )iml(i) =iml(i) +nx
2747: |                 if(im2(i).lt.1 )im2(i) =im2(i) +nx
 :                                                          } 参照するアドレスが 1 を下回る場合, nx に戻る
2755: |                 if(iml0(i).lt.1 )iml0(i) =iml0(i) +nx
2756: |                 if(iml1(i).lt.1 )iml1(i) =iml1(i) +nx
2757: V------           enddo
```

図 4.7-14 間接参照の格納値

各配列に格納されている値は, ループ変数 i に対して i+1(ip1), i+2(ip2), i-1(iml), i-2(im2) というように ip は i のプラス, im は i のマイナスを意味しており, 配列名の数字はプラスおよびマイナスの値を意味している. 配列 ip1～ip11, iml～iml1 に格納されている値を以下の表 4.7-1, 表 4.7-2 に示す.

表 4.7-1 ip1～ip11 の格納値

配列名	ip1	ip2	ip3	ip4	ip5	ip6	ip7	ip8	ip9	ip10	ip11
格納値	i+1	i+2	i+3	i+4	i+5	i+6	i+7	i+8	i+9	i+10	i+11

表 4.7-2 iml～iml1 の格納値

配列名	iml	im2	im3	im4	im5	im6	im7	im8	im9	iml0	iml1
格納値	i-1	i-2	i-3	i-4	i-5	i-6	i-7	i-8	i-9	i-10	i-11

また, 配列 ip1～ip11, iml～iml1 の値が間接参照となる配列の配列外を参照しないように, 配列 ip1～ip11 の格納値は上限値 nx を超えると 1 から参照し, 配列 iml～iml1 は格納値が 1 を下回ると nx から参照するようになっている. 配列 ip1～ip11, iml～iml1 の格納値を考慮すると, i の DO ループにおいては, 12～nx-11 の範囲であれば, ip1(i)を i+1, ip2(i)を i+2, iml(i)を i-1, im2(i)を i-2 に直接書き換えても配列外参照はおこらず, 間接参照を取り除くことが可能となる. そのため間接参照を取り除くために i の DO ループをループ分割する.

同様に配列 jp1～jp11, jml～jml1 の値も設定されているため, j の DO ループもループ分割する. ループ分割後のソースコードを図 4.7-15, ループ分割のイメージ図を図 4.7-16 に示す.

```
!ループ分割
112:  +-----> do k=nz_s, nz_e
113:  |+-----> do j=1, ny
114:  ||\---> do i=1, 11                    ①
 :
193:  ++\---  enddo ; enddo ; enddo

194:  +-----> do k=nz_s, nz_e
195:  |+-----> do j=1, 11
196:  ||\---> do i=12, nx-11                A
 :
275:  ++\---  enddo ; enddo ; enddo

276:  +-----> do k=nz_s, nz_e
277:  |+-----> do j=12, ny-11
278:  ||\---> do i=12, nx-11                B          ②
 :
357:  ++\---  enddo ; enddo ; enddo

358:  +-----> do k=nz_s, nz_e
359:  |+-----> do j=ny-10, ny
360:  ||\---> do i=12, nx-11                C
 :
439:  ++\---  enddo ; enddo ; enddo

440:  +-----> do k=nz_s, nz_e
441:  |+-----> do j=1, ny
442:  ||\---> do i=nx-10, nx                ③
 :
521:  ++\---  enddo ; enddo ; enddo
```

図 4.7-15 DO ループの分割

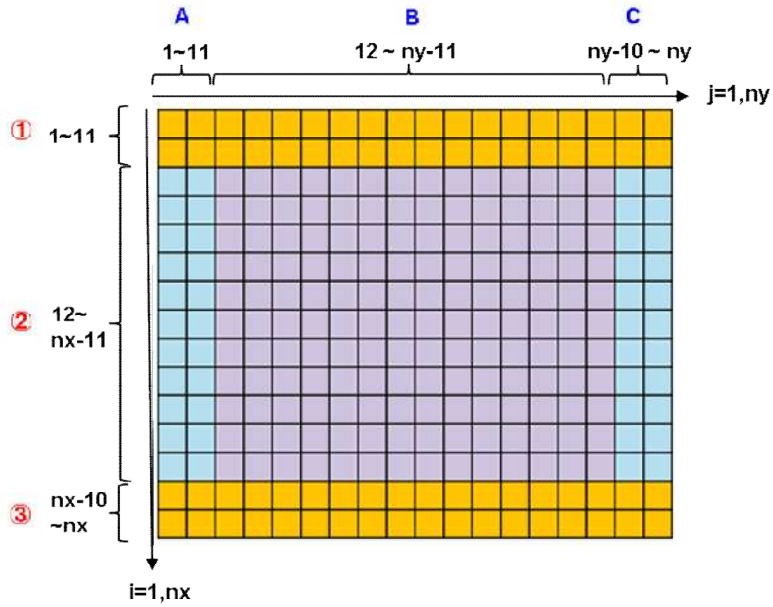


図 4.7-16 DO ループの分割イメージ

ループ分割後において, A とC の DO ループはループ長が 11 であるため最内側にループ入れ換えを行う. その後, 1, 3, A, C の DO ループは最内側のベクトル長が 11 となるのでループを展開する EXPAND, UNROLL 指示行を挿入する. また, B の DO には外側ループの展開を行う OUTERUNROLL 指示行を挿入する. ループ入れ換えおよびコンパイラ指示行を挿入したソースコードを図 4.7-17 に示す.

```
!ループ入れ換えとコンパイラ指示行の挿入
 112  +------>          do k=nz_s, nz_e
 113  |  V---->          do j =1, ny
 114  |  |              !cdir expand=11
 115  |  |*---->          do i =1, 11
  :
 194  |  V*----          enddo ;  enddo ;  enddo

 195  |  +------>        do k=nz_s, nz_e
 196  |  |  V---->        do i =12, nx-11
 197  |  |  |           !cdir expand=11
 198  |  |  |*---->        do j =1, 11
  :
 277  |  |  V*----        enddo ;  enddo ;  enddo

 278  |  +------>        do k=nz_s, nz_e
 279  |  |              !cdir outerunroll=8
 280  |  |+------>        do j =12, ny-11
 281  |  ||  V---->        do i =12, nx-11
  :
 360  |  ++  V----        enddo ;  enddo ;  enddo

 361  |  +------>        do k=nz_s, nz_e
 362  |  |  V---->        do i =12, nx-11
 363  |  |  |           !cdir unroll=11
 364  |  |  |*---->        do j =ny-10, ny
  :
 443  |  V*----          enddo ;  enddo ;  enddo
```

```
444:  +------>          do k=nz_s, nz_e
445:  |V----->          do j=1, ny
446:  |||           !cdir unroll=11
447:  |||*---->          do i=nx-10, nx
 :
526:  +V*----           enddo ; enddo ; enddo
```

図 4.7-17 DO ループの入れ換えとコンパイラ指示行の挿入

チューニング後の B の DO ループのソースコード全体を図 4.7-18 に示す.

```
!チューニング後
278:  +------>          do k=nz_s, nz_e
279:  |             !cdir outerunroll=8
280:  |+----->          do j=12, ny-11
281:  ||V---->          do i=12, nx-11
 :
303:  |||      A         diff_x=-(       +coef2_x11*t(i+11, j   , k  )
304:  |||               &                +coef2_x10*t(i+10, j   , k  )
305:  |||               &                +coef2_x9 *t(i+ 9, j   , k  )
306:  |||               &                +coef2_x8 *t(i+ 8, j   , k  )
307:  |||               &                +coef2_x7 *t(i+ 7, j   , k  )
308:  |||               &                +coef2_x6 *t(i+ 6, j   , k  )
309:  |||               &                +coef2_x5 *t(i+ 5, j   , k  )
310:  |||               &                +coef2_x4 *t(i+ 4, j   , k  )
311:  |||               &                +coef2_x3 *t(i+ 3, j   , k  )
312:  |||               &                +coef2_x2 *t(i+ 2, j   , k  )
313:  |||               &                +coef2_x1 *t(i+ 1, j   , k  )
314:  |||               &          +2.0d0*coef2_x0 *t(i    , j   , k  )
315:  |||               &                +coef2_x1 *t(i- 1, j   , k  )
316:  |||               &                +coef2_x2 *t(i- 2, j   , k  )
317:  |||               &                +coef2_x3 *t(i- 3, j   , k  )
318:  |||               &                +coef2_x4 *t(i- 4, j   , k  )
319:  |||               &                +coef2_x5 *t(i- 5, j   , k  )
320:  |||               &                +coef2_x6 *t(i- 6, j   , k  )
321:  |||               &                +coef2_x7 *t(i- 7, j   , k  )
322:  |||               &                +coef2_x8 *t(i- 8, j   , k  )
323:  |||               &                +coef2_x9 *t(i- 9, j   , k  )
324:  |||               &                +coef2_x10*t(i-10, j   , k  )
325:  |||               &                +coef2_x11*t(i-11, j   , k  ) )*alph0
327:  |||      A         diff_y=-(        coef2_y11*t(i   , j+11, k  )
328:  |||               &                +coef2_y10*t(i   , j+10, k  )
329:  |||               &                +coef2_y9 *t(i   , j+ 9, k  )
330:  |||               &                +coef2_y8 *t(i   , j+ 8, k  )
331:  |||               &                +coef2_y7 *t(i   , j+ 7, k  )
332:  |||               &                +coef2_y6 *t(i   , j+ 6, k  )
333:  |||               &                +coef2_y5 *t(i   , j+ 5, k  )
334:  |||               &                +coef2_y4 *t(i   , j+ 4, k  )
335:  |||               &                +coef2_y3 *t(i   , j+ 3, k  )
336:  |||               &                +coef2_y2 *t(i   , j+ 2, k  )
337:  |||               &                +coef2_y1 *t(i   , j+ 1, k  )
338:  |||               &          +2.0d0*coef2_y0 *t(i   , j    , k  )
339:  |||               &                +coef2_y1 *t(i   , j- 1, k  )
340:  |||               &                +coef2_y2 *t(i   , j- 2, k  )
341:  |||               &                +coef2_y3 *t(i   , j- 3, k  )
342:  |||               &                +coef2_y4 *t(i   , j- 4, k  )
343:  |||               &                +coef2_y5 *t(i   , j- 5, k  )
344:  |||               &                +coef2_y6 *t(i   , j- 6, k  )
345:  |||               &                +coef2_y7 *t(i   , j- 7, k  )
346:  |||               &                +coef2_y8 *t(i   , j- 8, k  )
347:  |||               &                +coef2_y9 *t(i   , j- 9, k  )
```

```
348:  |||                &                +coef2_y10*t(i  ,j-10,k  )
349:  |||                &                +coef2_y11*t(i  ,j-11,k  ))*alph0
:
360:  ++V---              enddo ; enddo ; enddo
```

図 4.7-18 複合事例 2 のチューニング後コード

(3) 性能分析

図 4.7-19 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	PROC.NAME
17	**73.223**(28.1)	4307.257	11676.6	5186.6	99.83	240.0	52.196	0.000	20.436	**3.408**	**41.798**	【チューニング前】
17	**10.157**(6.8)	597.450	78766.0	49072.7	99.80	235.7	10.126	0.001	0.004	**0.512**	**1.970**	【チューニング後】

図 4.7-19 複合事例 2 のチューニング前後 FTRACE 情報

ベクトルループ内の間接参照を取り除くことでバンクコンフリクト時間が 45 秒から 3 秒に減少することで演算効率が上がり, 実行時間は 73.2 秒から 10.2 秒に短縮することができた.

4.7.3. 事例 3

(1) チューニング方針

図 4.7-20 にチューニング前のコードを示す. チューニング前のコードは並列コンピュータ Express5800 で実行されていたコードであり, FFT 演算に Intel MKL ライブラリを使用している. SX-9 において MKL ライブラリは提供されていないため, FFT 演算に ASL ライブラリを使用するための変更を行った. チューニング前のコードでは使用する FFT 演算として, 複素引数型 3 次元 FFT 演算を行うことが指定されているので, ASL ライブラリの ZFC3BF を選択した.

次に, コードは問題サイズによって FFT 演算を行う配列サイズが変更されるため, サンプルコードを用いて演算を行う配列サイズ, 引数型, および並列数を変化させたときの実行時間の比較を行い, コード実行時に最適な FFT 演算ライブラリの選択と並列数の検討を行うこととした.

```
! FFT演算ライブラリ置き換え前
 35         status = DftiCreateDescriptor(handle, DFTI_DOUBLE, DFTI_COMPLEX, 3, length)
 37         status = DftiSetValue(handle, DFTI_PLACEMENT, DFTI_INPLACE)
 39         if(flag == 1) then
 40             scale = 1.0d0 / dble(nx*ny*nz)
 41             status = DftiSetValue(handle, DFTI_FORWARD_SCALE, scale)
 42             status = DftiCommitDescriptor(handle)
 43             status = DftiComputeForward(handle, src)
 44         else if(flag == -1) then
 45             status = DftiCommitDescriptor(handle)
 46             status = DftiComputeBackward(handle, src)
 47         endif
 48         status = DftiFreeDescriptor(handle)
```

図 4.7-20 FFT 演算ライブラリ置き換え前コード

(2) チューニング内容

<STEP1 ASL ライブラリへの置き換え>

FFT 演算部分を MKL ライブラリDfti*から, ASL ライブラリの ZFC3BF に置き換えを行った. 置き換え後のコードを図 4.7-21 に示す.

```
! FFT演算ライブラリ置き換え後
 13        integer :: LX, LY, LZ, nx, ny, nz, flag, IFAX(60), IERR
 14        real(8),allocatable :: TRIGS(:)
 16        complex(8),allocatable :: C(:,:,:), WK(:)
 30        LX=nx+1; LY=ny+1; LZ=nz+1
 31        allocate(C(LX,LY,LZ),TRIGS(2*(nx+ny+nz)),WK(LX*LY*LZ))
 32        C(:,:,:)=(0.0d0, 0.0d0)
158        C(1:nx, 1:ny, 1:nz)=src(1:nx, 1:ny, 1:nz)
159        scale=1.0d0/dble(nx*ny*nz)
161        call ZFC3BF(nx,ny,nz,C,LX,LY,LZ,flag,IFAX,TRIGS,WK,IERR)
162        if (flag.eq.-1) scale=1.0d0
163        src(1:nx, 1:ny, 1:nz)=C(1:nx, 1:ny, 1:nz)*scale
```

図 4.7-21 FFT 演算ライブラリ置き換え後コード

置き換え後のコードを SX-9 で実行し, 得られた FFT 演算を行うサブルーチン部分の FTRACE 情報を図 4.7-22 に示す. コードの並列化は行わず, 1CPU での実行を行った.

PROC.NAME	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
【複素引数型・シングルスレッド実行】												
	726	288.014(48.7)	396.714	72081.7	31509.8	99.82	256.0	287.967	0.022	0.025	39.086	85.355

図 4.7-22 ＜STEP1＞FFT 演算ライブラリ置き換え後の FTRACE 情報

<STEP2 実行並列数および配列引数型の検討>

問題サイズに対して最適な FFT 演算ライブラリを選択するために, サンプルコードにより3次元 FFT 演算に使用する配列サイズ, 実行並列数, および配列の引数型 (複素引数型・実数引数型) を変えて実行時間を比較した. 複素引数型の FFT 演算には HFC3BF, 実数引数型の FFT 演算には QFC3BF の各 ASL ライブラリを使用した. 配列サイズは現在のコードで用いられている 512×512×512 から, 2,048×2,048×2,048 まで変化させた. 3 次元 FFT 演算は, 複素引数型では ZFC3FB および ZFC3CL, 実数引数型では DFC3FB および DFC3CL の各 ASL ライブラリにより初期化を行った後, 順変換と逆変換の組み合わせをそれぞれ 10 回行いその実行時間を比較した. また, 実数引数型の FFT 演算ライブラリでは配列データの複素数・実数変換のための演算時間も実行時間に含む.

複素引数型の FFT演算ライブラリ HCF3BF を使用したときの演算時間割合を図 4.7-23 に, 複素引数型の FFT演算ライブラリ QFC3BF を使用したときの演算時間割合を図 4.7-24 に示す. どちらの表も実行並列数 1 のときの演算時間に対する割合を示しているので, 値が 100%よりも小さい箇所は並列実行の効果が得られた事例を示す.

	並列数				
n	1	2	4	8	16
512	100%	55%	33%	23%	33%
1024	100%	54%	28%	17%	13%
2048	100%	49%	27%	17%	11%

図 4.7-23 シングル実行時間に対する実行時間割合（複素引数型）

	並列数				
n	1	2	4	8	16
512	100%	53%	30%	18%	13%
1024	100%	51%	26%	13%	8%
2048	100%	50%	25%	13%	7%

図 4.7-24 シングル実行時間に対する実行時間割合（実数引数型）

これらの結果より，いずれの問題サイズにおいても複素引数型・実数引数型とも並列数を 16 として実行した場合の実行時間が最も短い結果となった．

次に，引数型の違いによる FFT 演算ライブラリの演算時間を比較するために，複素引数型に対する実数引数型の実行時間割合を図 4.7-25 に示す．表の値が 100%よりも小さい箇所は，実数引数型の FFT 演算ライブラリを用いた方が実行時間が短いことを示す．

	並列数				
n	1	2	4	8	16
512	106%	101%	93%	82%	43%
1024	101%	96%	92%	81%	62%
2048	97%	100%	90%	76%	66%

図 4.7-25 複素引数型に対する実数引数型 FFT 演算ライブラリの実行時間割合

この結果より，今回の問題サイズ 512×512×512 では，並列数が 4 以上の実行の場合，実数引数型 FFT 演算ライブラリの方が演算時間が短いことが分かった．

以上の結果より今回の問題サイズでは，実数引数型 FFT 演算ライブラリを用いて並列数を 16 で実行することが最適であると判断した．置き換えたコードを図 4.7-26 に示す．

```
! 問題サイズに最適な FFT 演算ライブラリ置き換え後
 13        integer :: LX, LY, LZ, nx, ny, nz, flag, IFAX(60), NT, IERR
 14        real(8),allocatable :: TRIGS(:)
 15        real(8),allocatable :: CR(:,:,:),CI(:,:,:),WK(:)
 30        LX=nx+1; LY=ny+1; LZ=nz+1 ; NT=16
 31        allocate(C(LX,LY,LZ),TRIGS(2*(nx+ny+nz)),WK(LX*LY*LZ))
 32        CR(:,:)=(0.0d0, 0.0d0); CI(:,:)=(0.0d0, 0.0d0)
137        CR(1:nx,1:ny,1:nz)=dble(src(1:nx,1:ny,1:nz))
138        CI(1:nx,1:ny,1:nz)=aimag(src(1:nx,1:ny,1:nz))
159        scale=1.0d0/dble(nx*ny*nz)
160        call CFFT3BF(nx,ny,nz,CR,CI,LX,LY,LZ,flag,IFAX,TRIGS,WK,NT,IERR)
161        if (flag.eq.-1) scale=1.0d0
162        src(1:nx,1:ny,1:nz)=dcmplx(CR(1:nx,1:ny,1:nz),CI(1:nx,1:ny,1:nz)) * scale
```

図 4.7-26 ＜STEP2＞問題サイズに最適な FFT 演算ライブラリ置き換え後コード

(3) 性能分析

図 4.7-27 に，複素引数型・シングル実行の FFT 演算ライブラリと，実数引数型・16 並列実行の FFT 演算ライブラリで実行した際の FTRACE 情報を示す．

PROC.NAME	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
【複素引数型・シングル実行】												
	726	288.014(48.7)	396.714	72081.7	31509.8	99.82	256.0	287.967	0.022	0.025	39.086	85.355
【実数引数型・16並列実行】												
	726	35.816(17.5)	49.333	48099.7	15837.6	99.64	254.4	35.018	0.031	0.096	3.584	14.764

図 4.7-27 問題サイズに最適な FFT 演算ライブラリ選択前後の FTRACE 情報

今回のコードで用いられる問題サイズに最適な FFT ライブラリと並列数を選択することで，SX-9 での実行時間を 288.0 秒から 35.8 秒に短縮することができた．

今回の問題サイズは 512×512×512 であり，最適な FFT 関数と並列数として，実数引数型・16 並列実行を選択した．今後の研究では問題サイズが拡張され，2,048×2,048×2,048 での実行が予定されている．この問題サイズでの実行の際も図 4.7-24 および図 4.7-25 の結果より，実数引数型・16 並列実行での実行が最適であると判断される．

5. MPI 化による高速化

スーパーコンピューティング研究部　小林広明　江川隆輔　小松一彦　岡部公起
情報部情報基盤課　大泉健治　小野敏　山下毅　佐々木大輔　森谷友映　齋藤敦子
日本電気株式会社　撫佐昭裕　松岡浩司　渡部修
NEC ソリューションイノベータ株式会社　曽我隆　山口健太

5.1. MPI 概要

単一のプロセッサ(あるいはコア. 以下総称してプロセッサとする)のみでプログラムを実行することを逐次実行という. それに対して複数のプロセッサを並列に動作させることでプログラムを実行することを並列実行という. 並列実行には以下の利点が挙げられる.

・プログラムの実行時間の短縮.

・大容量のメモリ空間を必要とする計算の実行

図 5.1-1 に並列化の考え方を示す. 逐次実行時に「処理 A」+「処理 B」+「処理 C」を処理する時間(経過時間)を要するプログラムの実行に対して,「処理 B」の実行を複数に分散して並列に行う(図では 4 並列で実行)ことにより,「処理 B」を処理する時間の短縮を図る.「処理 A」と「処理 C」を処理する時間は逐次実行時と同じであるが,「処理 B」を処理する時間が短縮されたことにより, 結果として全体の処理時間(経過時間)を逐次実行より短くすることが並列実行の効果である. このように「処理 B」を複数のプロセッサに分散して並列に実行することを並列化という.

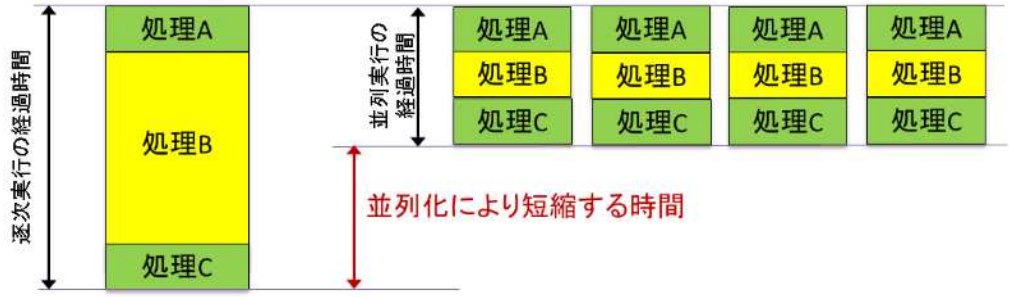


図 5.1-1 並列化の考え方

図 5.1-2 に並列化による実行時間短縮のイメージを示す. プロセッサ N 台を用いて「N 並列における実行時間を 1/N にする」ことが並列実行の理想であるが, 並列化に伴い逐次実行時には必要のない処理が並列実行時に加わる. これを並列化によるオーバーヘッドという. 並列化によるオーバーヘッドにはプロセッサ間の足並みを揃える(同期を取る)処理やプロセッサ間の情報交換(データ転送)などが挙げられる. 並列実行の理想である「N 並列における実行時間を 1/N」に近づけるためには, 並列化によるオーバーヘッドの時間をできるだけ短くすることが必要になる.

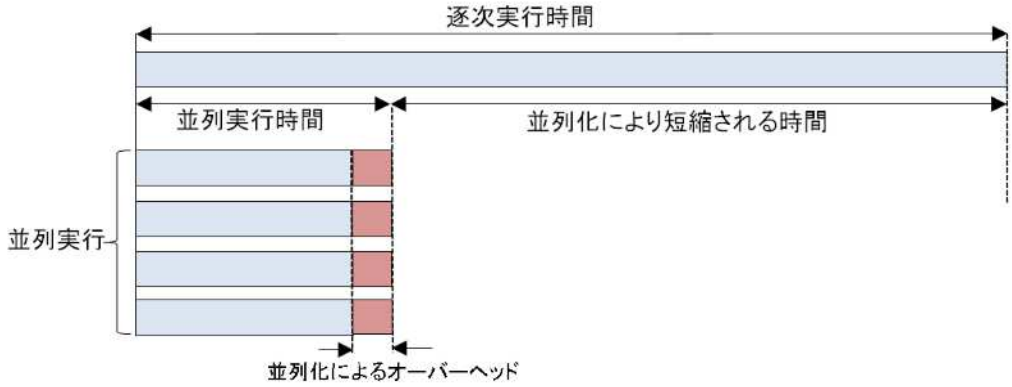


図 5.1-2 並列化による時間の短縮

図 5.1-3 にプログラムを実行するハードウェアの構成を示す．並列化のモデルは，ハードウェアの構成により大きく二つに分類される．図 5.1-3(a)に示す共有メモリ環境では，各プロセッサは同一のメモリ空間をアクセスすることが可能である．このような環境では，各プロセッサが同じメモリアドレスにあるデータを更新や参照を行う際の整合性を保つことさえ確保すれば容易に並列実行が可能になる．そのためコンパイラの自動並列化機能や指示行の挿入レベルで比較的簡単に並列化が可能になる．しかしメモリ空間の大きさに限りがあり，プログラムが扱える物理空間に制限が生じてしまうため大規模シミュレーションの実行が難しくなる．

図 5.1-3(b)に示す分散メモリ環境では，プロセッサごとに担当するメモリ空間が独立しており(このプロセッサとメモリの組み合わせをノードと定義する)，ノードを増やすことにより理論上はメモリ空間を限りなく増やすことが可能になる．しかし分散メモリ環境では異なるプロセッサが管理するメモリ空間のデータを別のプロセッサがアクセスするには，ネットワークを介してデータを通信することが必要である．またプログラムが計算する物理領域をプロセッサに割り当てるという操作が必要になる．できるだけ通信が発生しないように，通信が発生したとしても回数やデータ通信量を最少になるように領域を分割することが必要であり，コンパイラによる自動並列化が難しい部分である．

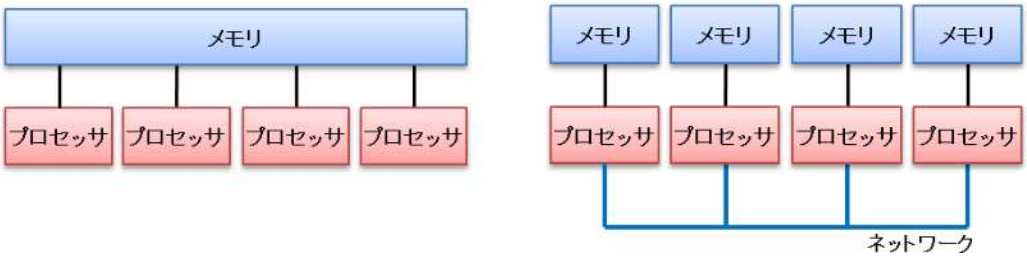


図 5.1-3(a) 共有メモリ　　図 5.1-3(b) 分散メモリ

図 5.1-3 プログラムを実行するハードウェアの構成

分散メモリ環境におけるプログラムの並列実行を行う方法として，MPI(Message Passing Interface)が挙げられる．MPI には次の特徴がある．

① 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，複数のプロセッサ間でデータのやり取りを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．

② FORTRAN や C 言語プログラムから呼び出すサブプログラムのライブラリである．

③ ポータビリティに優れており，様々な MPI 実装環境でソース修正なく使用できる．

④ MPI を利用することで複数のノードを利用でき，使用するメモリ空間を広げることが可能である．

⑤ 物理空間の分割やデータ通信処理の実装など，ユーザの負担は比較的大きい．

以下，MPI を利用することで大規模並列シミュレーションを実現するための手順や並列化の効果を最大限にする最適化手法の事例を説明する．

5.2 MPI 化による大規模並列シミュレーションの実現

近年，高性能計算機(HPC)システムを活用したシミュレーションにおいては，シミュレーション対象の大規模化・高精度化が進んでおり，それに伴い，アプリケーションの演算量および使用メモリ量が飛躍的に増加してきている．また，同時にシミュレーション結果を得るまでの時間短縮も求められている．このような要求を満たすため HPC システムの性能も年々向上し続けており，それを実現するためにより多くの計算機資源の活用が必要となる．東北大学サイバーサイエンスセンターにて導入している SX-9 は，各ノードについては共有メモリ型計算機であり，システム全体としては，各ノード間を高速ネットワーク(インターノードクロスバースイッチ:IXS)で接続した共有分散メモリ型計算機である．この東北大学サイバーサイエンスセンターの SX-9 において，MPI による並列化をアプリケーションに適用することにより，マルチノードシステムとして計算機資源を有効活用した事例を下記に示す．

本事例で紹介するアプリケーションは全球地震波伝搬シミュレーションを行うプログラムである．当初は自動並列化機能を用いた並列化によりノード内での計算を行っていたが，この時点では長周期地震波での計算にとどまっていた．さらに現実的なシミュレーションを行うためには，できるだけ短周期に近づけていく必要があるが，周期を短くすると，空間格子間隔および時間格子間隔を細かくする必要があるため，計算量および使用メモリ容量も指数関数的に増加する．従って短周期のシミュレーションを行うためには，より多くの計算機資源を活用した計算が必要となる．そこで，プログラムに MPI による並列化を適用し，マルチノードでの実行を可能とすることにより，短周期地震波におけるシミュレーションを実現した．この計算に使用した問題規模は，これまで計算対象としていた問題規模と比較して，約 2500 倍の演算量かつ約 200 倍の使用メモリ容量を必要とする．

図 5.2-1 に MPI による並列化を適用した全球地震波伝搬シミュレーションプログラムを SX-9 マルチノードシステムで実行したときの並列効果を示す．

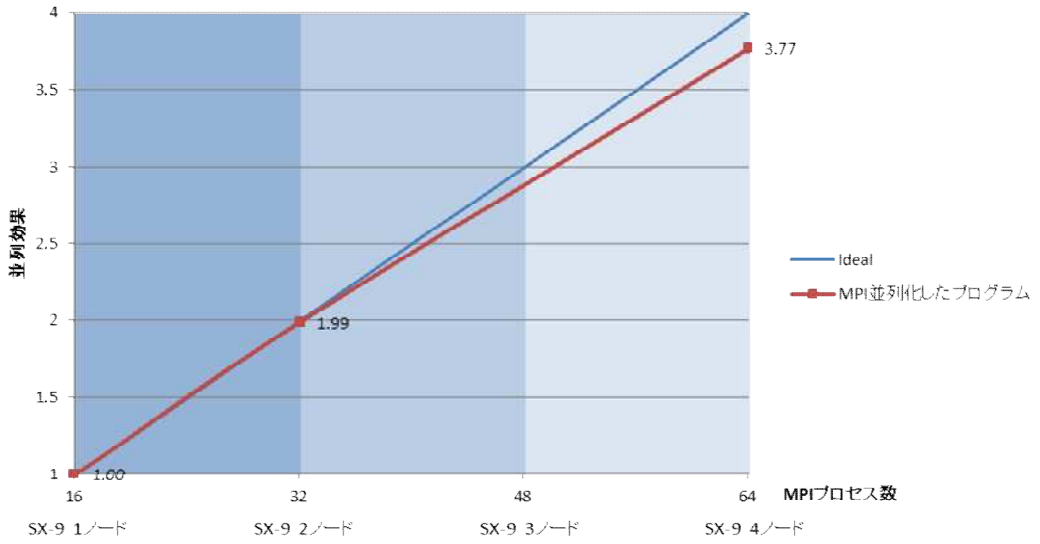


図 5.2-1 MPI 並列化したプログラムの並列効果

図 5.2-1 が示すように，1 ノードにおける実行時間を 1 とした場合，2 ノードでの実行では 1.99 倍，4 ノードでの実行では 3.77 倍と高い並列効果を得ており，この場合の並列化率は 99.87%である.

この事例が示すように，MPI を用いてアプリケーションの並列化を行うことにより，より多くの計算機資源を活用した計算を行うことができるようになり，これまで以上に大規模・高精度なシミュレーションが可能となる.

5.3. MPI 化の基本

本章では，MPI を用いてプログラムの並列化を行う手順を示し，それぞれの手順の中で考慮しなければならない事柄について説明する．

5.3.1. MPI 化の手順

5.1 で述べたように，プログラムを MPI 化することにより「プログラムの実行時間の短縮」や「利用するメモリ空間の拡大」という効果を得ることができる．しかしながら MPI 化は共有メモリ並列のようにコンパイラが自動で行うことは現状ではまだ難しく，領域の分割やデータ転送処理の実装は開発者自身で実施する必要がある．

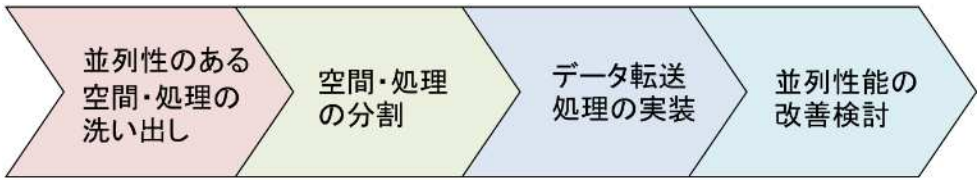


図 5.3-1 MPI 化の手順

図 5.3-1 に MPI 化の手順を示す．MPI 化の最初の作業はプログラムの中のどの部分に並列性があるのか検討することである．この並列性とはプログラムを複数のプロセッサで並列に実行しても計算結果が正しく得られることを意味する．時間積分を行うような処理は，前の繰り返しの結果を受けて次の繰り返しの計算が行われるというような依存関係が存在する．そのため処理を分割して並列に実行すると計算結果が逐次で実行する場合と異なるので，この処理には並列性がないと言える．物理空間を扱うようなプログラムでは，物理空間を複数の空間に分割して実行しても逐次実行と同様の計算結果を得られる場合が多く，この場合は物理空間に並列性があると言える．また複数の事象を扱うプログラムでは，事象ごとに関連がない(依存する関係がない)場合，事象を複数のプロセッサで分担して実行することが可能になる．この場合事象に並列性があると言える．

次に並列性のある物理空間や事象を分割してプロセッサに割り当てる．ここで注意しなければならないのは，物理空間の大きさを均等に分割するのではなく，演算量が均等になるように分割しなければならないことである．これは事象を分割する場合も同様で，事象の数を均等に割り振るのではなく，事象内の演算量が均等になるように割り振らなければならない．プロセッサが分担する演算量が不均一な場合，演算が早く終わったプロセッサは遅いプロセッサを待たなければならず，プロセッサ数に応じた効果(N 台のプロセッサで実行する場合の最大の効果は実行時間が 1/N になること)が得られないことになる．この演算量を均一にすることによりロードバランスが均一となる．

次にプロセッサ間のデータ転送処理の実装を検討する．物理空間を扱うプログラムでは隣接するプロセッサが担当する空間に存在するデータをプロセッサが自分のメモリ空間上にあるデータの様に直接アクセスすることはできないため，データ転送という方法を用いる．プロセッサ間のネットワークを介して通信することで，隣接プロセッサが担当するデータを自分のメモリ空間上に格納することで，プロセッサはデータのアクセスが可能になる．MPI では様々なデータ転送方法を提供している．特定のプロセッサ間でのデータ転送を行う場合は一対一通信と呼ばれる方法を用いる．また，あるプロセッサの集団

を定義(コミュニケータという単位で管理)しておき，同じコミュニケータに属するすべてのプロセッサが参加するデータ転送方法を提供している．これを集団通信と呼ぶ．集団通信の中にはあるプロセッサのデータを一斉に通知するMPI_Bcastや，同じコミュニケータ内のプロセッサが有するデータの総和や最大値・最小値を求めるMPI_Reduceなどがある．

一対一通信(MPI_Send/MPI_Recv)があれば，どのようなデータ通信処理であっても記述することが可能であるが，前述のような複数のプロセッサが参加するデータ通信処理を一対一通信だけで記述するのは煩雑になる．また集団通信の多くは用意しているベンダー側で最適化(できるだけデータ転送時間を短くする仕組みの提供)がなされているので，集団通信が使える場合は集団通信を利用する．しかしながらデータ通信処理は本来の逐次処理実行では必要のない処理であり，データ通信にかかる処理時間がそのままオーバーヘッドとなる．N台のプロセッサを使用して実行時間を1/Nにするためには，データ通信処理の時間は0にしなければならないのであるが，データ通信処理を完全になくしてしまうことは不可能である．そこでN並列時の実行時間を1/N時間に近づけるためには，出来るだけデータ転送処理回数を削減することと，データ転送量を削減することを検討しなければならない．プロセッサに割り当てる演算量を均一にすることと，データ転送処理回数や転送量の削減の検討が並列化により効果を最大限にする(1/N時間に近づける)ために必要になる．

5.3.2 領域分割

プログラムが計算する物理空間に並列性がある場合，領域分割法と呼ばれる物理空間の分割方法を用いてプロセッサに物理空間を分担させる．領域分割法にはブロック分割とサイクリック分割の二つの手法がある．図 5.3-2 にブロック分割の例を示す．ブロック分割は物理空間をプロセッサ数で大きく分割して割り当てる方法である．ブロック分割の利点はプロセッサが分担する空間の隣接面は最少になるのでデータ転送処理が抑えられる点である．しかしながら物理空間を大きく分割するため，プロセッサが担当する空間が大きくなり，場所によって演算の条件が異なるようなプログラム(たとえば空間の中の形状や材質が均一でなく，異なる演算を実行しなくてはならないような場合)はロードバランスの不均一が発生する．図 5.3-3 にサイクリック分割の例を示す．サイクリック分割は物理空間を細分化してプロセッサに割り当てる(したがってプロセッサは複数に分けられた空間を担当する)方法である．サイクリック分割の利点は，空間の場所によって演算量が異なっても，空間を細分化してプロセッサに割り当てるため，演算量が均される可能性が大きいので，空間内の演算量が不均一な場合に適した分割方法である．しかしながらプロセッサが担当する空間の隣接面が大きくなるため，データ転送処理の回数やデータ転送量が多くなる可能性がある．このようにブロック分割とサイクリック分割は長短があるため，ユーザは最適な分割方法を選択する必要がある．

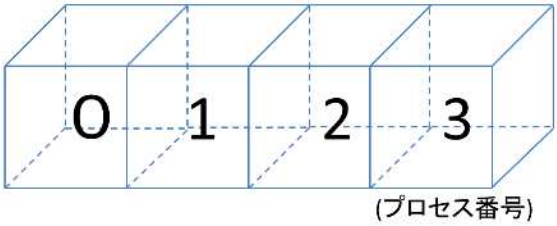


図 5.3-2 ブロック分割

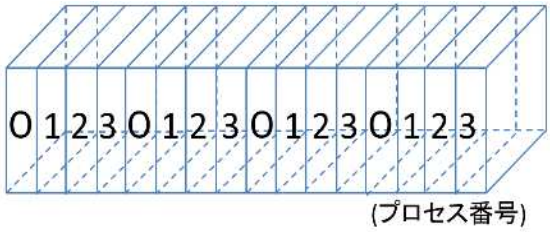


図 5.3-3 サイクリック分割

図 5.3-4 に配列の分割方向の例を示す．この例では空間を 4 プロセッサに分割するものとする．m×n の大きさを持つ配列 A は FORTRAN プログラムでは A(m,n)と定義され，メモリ空間上で A(1,1),A(2,1),…A(m,1),A(1,2),…と一次元目のアドレスで連続して格納される．中央の図は二次元目の n を分割する例である．この例では各プロセッサに割り当てられる配列 A はメモリ空間上で連続に分割されることになる(m×$^{n}/_{4}$の大きさ)．右の図は一元目の m を分割する例である．この例では各プロセッサに割り当てられる配列 A はメモリ空間上細かく分断されて割り当てられる($^{m}/_{4}$まで連続で m×$^{3}/_{4}$飛びに格納)．一般的にメモリアドレスができるだけ連続になるように分割する方が並列化によるオーバーヘッドは少なくなると考えられている．しかしながら配列 B(100,100,100)を三次元目だけで分割する場合，100分割までしかできないことになる．つまり使用できるプロセッサ数は100個となる．より多くのプロセッサを用いる並列化を行いたい場合は，二次元目と三次元目の両方を分割の対象とする(二次元分割)ことで最大 10,000 個のプロセッサを利用することが可能になる．

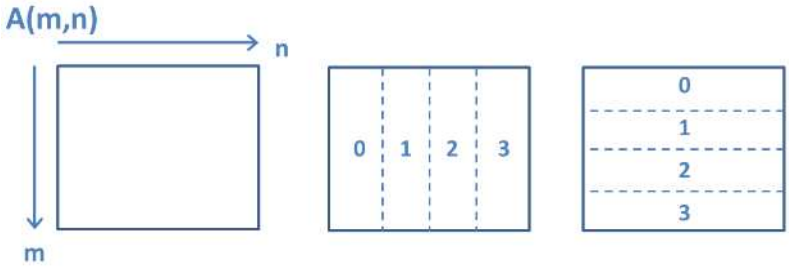


図 5.3-4 配列の分割方向

5.3.3. データ転送

異なるプロセッサが担当している空間に存在するデータを参照する必要がある場合，データ転送により自分の担当する空間にデータを移送する．MPI では次のようなデータ転送方法を提供している．

- 一対一通信(特定のプロセッサ間でデータ通信を行う)
- 集団通信(同じグループに属する複数のプロセッサ内で同時にデータ通信を行う)
- 単方向通信(相手プロセッサに関係なくデータを更新・参照可能な通信)

ユーザは最適な通信方法を選択してデータ転送処理を実装する必要がある．以下，一対一通信と集団通信について，簡単な例を挙げて実装方法を説明する．

(1) 一対一通信

差分式の計算では配列の±1や±2の要素を参照しながら計算を実行する．図 5.3-5 に差分式の例を示す．この例では配列 a(i)の計算に配列 b(i)と b(i-1)を参照する．配列 b(i)はプロセッサが担当する空間に存在するので問題ないが，配列 b(i-1)は別のプロセッサが担当する可能性がある．別のプロセッサの処理の過程で配列 b(i-1)が更新される場合，更新された内容をデータ転送で取得する必要がある．

```
do i=1,100
    a(i)= a(i) + b(i) + b(i-1)
end do
```

図 5.3-5 差分式の例

図 5.3-6 の境界要素の参照の例が示すように，25 番目の要素はプロセッサ 0 が担当しており，プロセッサ 1 が演算に使用する場合はプロセッサ 0 からデータ転送処理により25 番目の要素のデータを取得しなければならない．MPI の一対一通信を用いて，データ転送処理を行う．

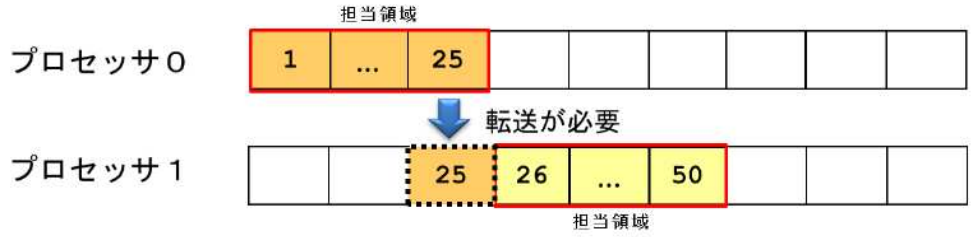


図 5.3-6 境界要素の参照

図 5.3-7 に一対一通信の例を示す．本例では非同期型の一対一通信である MPI_Isend/ MPI_Irecv を使用しているので，MPI_Wait を用いて同期を取っている．myrank に MPI のプロセス番号を識別するランク番号が格納されている．ランクの 0 番は送信する相手がいないので受信処理を行わない．同様にランクの nprocs-1 番(nprocs は総プロセス数)は受信する相手がいないので送信処理を行わない．itag は送受信処理を識別するタグである(ここでは itag=1 とする)．MPI_Wait の処理が終了するまで転送が完了していないので，配列 b(i-1)に対する更新や参照はしてはならない．MPI_Wait の引数である status は送信プロセスの情報が格納されるが，特に送受信処理に問題がなければ内容を確認する必要はない．

```
17:      if(myrank.ne.0) then
18:        call MPI_IRECV(b,m*n,MPI_REAL8,myrank-1,
19:     +                 itag,MPI_COMM_WORLD,ireq1,ierr)
20:      endif
21:      if(myrank.ne.nprocs-1) then
22:        call MPI_ISEND(a,m*n,MPI_REAL8,myrank+1,
23:     +                 itag,MPI_COMM_WORLD,ireq2,ierr)
24:      endif
25:      if(myrank.ne.0)        call MPI_WAIT(ireq1,status,ierr)
26:      if(myrank.ne.nprocs-1) call MPI_WAIT(ireq2,status,ierr)
```

図 5.3-7 一対一通信の例

図 5.3-7が示す一対一通信の例では非同期型の一対一通信を使用したが，図 5.3-8が示す同期型通信を用いる一対一通信の例では，図 5.3-9 が示す同期型の一対一通信の流れのように，ランク1 番の送信処理(③)はランク0 番からの受信処理(②)が完了しないと始められず，ランク2 番の送信処理(⑤)はランク 1 番からの受信処理(④)が完了しないと始められないというように，送受信処理に依存関係が生じるため並列に実行されない(逐次処理で実行される)．SX-9 の 1 ノード16CPU

を用いて，2GByte のデータを隣接するプロセッサに転送する実験を行ったところ，図 5.3-8 が示すような同期型の MPI_Send/MPI_Recv を用いた場合 0.36 秒必要なのに対して，図 5.3-7 が示すような非同期型の MPI_Isend/MPI_Irecv を用いると0.06秒で完了した．このように実装の方法により転送処理時間に影響を及ぼす可能性はあるので注意が必要である(同期型の一対一通信でも逐次的転送処理にならない実装方法を行うことは可能である)．

```
16:                 if(myrank.ne.0) then
17:                   call MPI_RECV(b,m*n,MPI_REAL8,myrank-1,
18:                  +              itag,MPI_COMM_WORLD,status,ierr)
19:                 endif
20:                 if(myrank.ne.nprocs-1) then
21:                   call MPI_SEND(a,m*n,MPI_REAL8,myrank+1,
22:                  +              itag,MPI_COMM_WORLD,ierr)
23:                 endif
```

図 5.3-8 同期型通信を用いる一対一通信の例

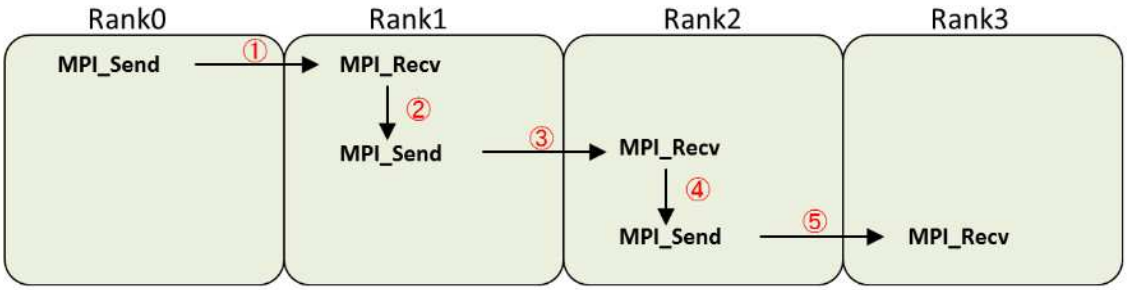


図 5.3-9 同期型の一対一通信の流れ

(2) 集団通信

ある演算の結果から次の処理を決定する(たとえば演算結果の収束判定)場合，全プロセッサから結果を集約して，次の処理内容を全プロセスへ通知するような転送処理が必要になる．図 5.3-10 に収束判定を行うループの例を示す．変数 sum の結果を判定してループを継続もしくは終了の処理を行う．

```
11:  +-----> do it=1,m
12:  |W----->  do j=1,n
13:  |||*---->   do i=1,n
14:  |||          sum= sum+ p(it)*a(i,j) - b(i,j)*c(i,j)
15:  |||*----    enddo
16:  |W----     enddo
17:  |          if(abs(sum).ge.eps) exit
18:  +-----     enddo
```

図 5.3-10 収束判定を行うループ

ループを分割して実行する場合，各プロセッサには部分和である変数 sum を持つことになる．この部分和では収束の判定を行うことはできないので，全プロセッサからsumを集約し，集約した結果を全プロセッサに返して収束判定を行う必要がある．このような場合，MPI 集団通信のリダクション演

算を行うMPI_AllReduce を用いる. 図 5.3-11 に MPI_AllReduce を用いる収束判定の例を示す. 変数 sum2 に全プロセッサから集約した結果が格納される. 演算の識別子はMPI_SUM を用いることで総和演算を行うことを指定する. MPI_Reduce を用いて特定のランクのみで実行すると, 収束後の処理が正しく実行されないので注意が必要である. 図 5.3-11 の例ではコミュニケータとして MPI_COMM_WORLD を使用している. これは全プロセッサが参加するコミュニケータであるが, 特定のプロセッサ内で集団通信を行いたい場合は新たなコミュニケータを定義しておく必要がある.

```
25: +-----> do it=1,m
26: |W----> do j=ist,ied
27: ||*----> do i=1,n
28: ||| sum= sum+ p(it)*a(i,j) - b(i,j)*c(i,j)
29: ||*---- enddo
30: |W---- enddo
31: | call MPI_ALLREDUCE(sum,sum2,1,MPI_REAL8,MPI_SUM,
32: | + MPI_COMM_WORLD,ierr)
33: | if(abs(sum2).ge.eps) exit
34: +----- enddo
```

図 5.3-11 MPI_AllReduce を用いる収束判定の例

5.4. MPI プログラムの高速化

並列化による実行時間の短縮で最大の効果(理想)はＮプロセッサによる並列で時間が 1/Ｎになることであるが，MPI を用いる並列プログラムでは

- プロセッサに割り当てる処理量を均一にする
- データ転送回数またはデータ転送量(その両方)の削減

の 2 点が重要になる．特にデータ転送処理に関しては，できればデータ転送行わないことが最善であるが，正しい結果を得るためにはデータ転送処理は不可欠である．しかしながらデータ転送方法を見直すことでデータ転送処理時間の短縮が可能になる場合がある．

以下，MPI プログラムの高速化の事例を挙げて，どのように改善を図ったのか説明する．

5.4.1. 事例 1(処理量の均一化)

(1) チューニング方針

図 5.4-1 に外側ループで内側ループの長さが決まるループの例を示す．本例では並列化の対象とするループは 14 行目から 18 行目のループである．実際の演算量を決める最内のループのループ長は，外側のループのループ変数(j)で決まる．図 5.4-2 に外側ループで内側ループの長さが決まるループの処理量を示す．各プロセッサに割り当てられる処理量に偏りがあり，ランク0を担当するプロセッサはランク3を担当するプロセッサの７倍の処理を実行しなければならない．このようにプロセッサに割り当てる処理量に偏りがあるため，4 台のプロセッサで時間は 7/16(理想は 1/4)にしかならない．分割方法を変更してプロセッサへ割り当てる演算量の均一化を検討する．

```
13:  +-----> do it=1,m
14:  |+----->  do j=1,n
15:  ||V--->    do i=1,n-j+1
16:  |||         sum= sum+ p(it)*a(i,j) - b(i,j)*c(i,j)
17:  ||V---     enddo
18:  |+-----   enddo
19:  +-----  enddo
```

図 5.4-1 外側ループで内側ループの長さが決まるループ

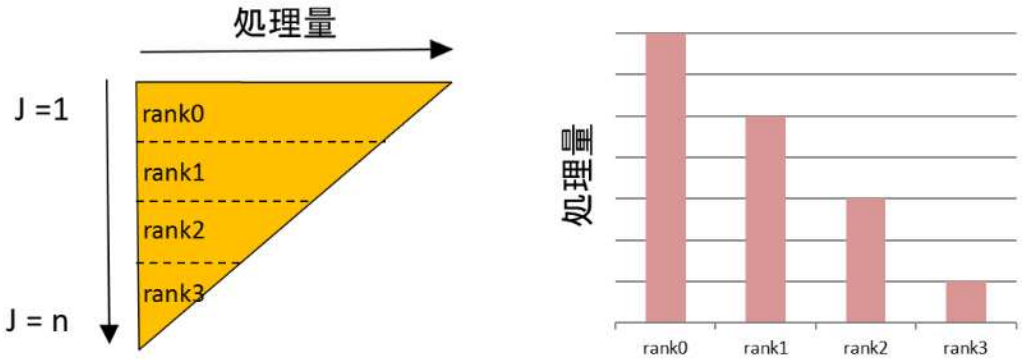


図 5.4-2 外側ループで内側ループの長さが決まるループの処理量

2) チューニング内容

図 5.4-3 に外側ループの分割数を増やした場合の処理量の例を示す. 分割数を 2 倍に増やして処理が終わったプロセッサに次の計算領域を割り当てるようにすることで, 全プロセッサに割り当てる処理量を均一にすることが可能になる.

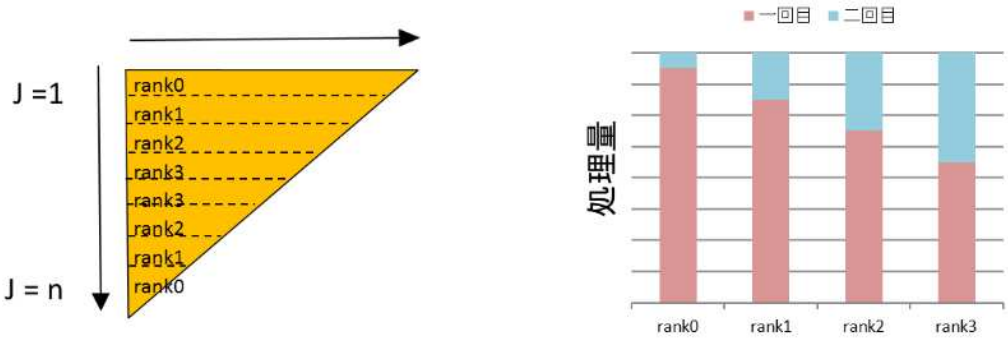


図 5.4-3 外側ループの分割数を増やした場合の処理量

3) 性能分析

図 5.4-4 にループ分割数を変更した場合の FTRACE 情報を示す. 20.1 秒の処理時間が 12.3 秒に改善している. ループをプロセッサ数の 4 に分割した場合の VECTOR TIME にはランクごとに偏りが見られるが, 8 分割して割り当てた場合の VECTOR TIME はほぼ均一になっている. この例では分割数を倍にすることで処理量の均一化を図ることが可能になったが, ループ中の処理の偏りが大きい場合はサイクリック分割によるループの分割方法が有効の場合がある.

PROC. NAME	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
(修正前)												
loop	4	80.597(45.1)	20149.370	23941.8	13427.6	99.16	249.3	54.646	0.000	0.001	0.208	31.211
0.0	1	20.149	20149.443	41458.2	23419.2	99.42	253.1	20.149	0.000	0.000	0.057	10.245
0.1	1	20.149	20149.429	29779.7	16758.1	99.28	251.1	15.865	0.000	0.000	0.072	9.092
0.2	1	20.149	20149.429	18101.2	10097.0	98.97	246.5	11.620	0.000	0.000	0.045	6.919
0.3	1	20.149	20149.178	6427.7	3435.9	97.47	226.6	7.012	0.000	0.000	0.023	4.955
(修正後)												
loop	4	49.040(42.4)	12260.064	39240.0	22068.2	99.40	251.0	48.768	0.000	0.000	0.237	26.858
0.0	1	12.260	12260.056	39241.7	22068.2	99.39	251.0	12.092	0.000	0.000	0.050	6.462
0.1	1	12.260	12260.066	39238.9	22068.2	99.40	251.1	12.260	0.000	0.000	0.061	6.795
0.2	1	12.260	12260.073	39240.0	22068.2	99.40	251.0	12.190	0.000	0.000	0.065	6.840
0.3	1	12.260	12260.061	39239.5	22068.2	99.40	251.1	12.226	0.000	0.000	0.061	6.772

図 5.4-4 ループ分割数を変更した FTRACE 情報

図 5.4-5 に外側ループのサイクリック分割の例を示す. この例では外側ループをプロセッサ数飛びで実行することでサイクリック実行しているが, 処理を予めプロセッサに割り当てるため, ラウンドロビン・スケジューリングのように処理が先に終わったプロセッサに次の処理を割り当てることはできないため, この例では図 5.4-3 のように処理量は均一にならない.

```
 23: +-----> do it=1,m
 24: |+-----> do j=1+myrank,n,nprocs
 25: ||V---> do i=1,n-j+1
 26: ||| A sum= sum+ p(it)*a(i,j) - b(i,j)*c(i,j)
 27: ||V--- enddo
 28: |+----- endd
 29: +----- enddo
```

図 5.4-5 外側ループのサイクリック分割

このようにプロセッサに割り当てる処理量を均一にすることで並列性能の向上を図る例を示したが，MPI プログラムの処理時間の計測を正しく行わないと各プロセッサが担当している処理量を把握することができない．図 5.4-6 にタイマーの前に同期を取る場合の例を示す．同期を取ることで全プロセッサの処理の終了を待ち合わせるため，タイマーの値は最も遅いプロセッサの時間(この時間が並列実行時のループの処理時間)となる．しかしこの測定方法ではプロセッサ毎の正しい処理時間を計測することはできない

```
 25:                call MPI_BARRIER(MPI_COMM_WORLD,ierr)
 26:                t1=MPI_WTIME()
 27:                sum=0.0d0
 28: +------>       do it=1,m
 29: |+------>      do j=ist,ied
 30: ||V------>     do i=1,j
 31: |||    A       sum = sum + p(it)*a(i,j) - b(i,j)*c(i,j)
 32: ||V----        enddo
 33: |+-----        enddo
 34: +------        enddo
 35:                call MPI_BARRIER(MPI_COMM_WORLD,ierr)
 36:                t2=MPI_WTIME()
 37:                write(6,6000) myrank,t2-t1
 38:             6000 format('Myrank = ',i2,' Time = ',f10.3,' sec')
% mpirun -np 4 ./a.out
Myrank =  1 Time =     19.733 sec
Myrank =  0 Time =     19.733 sec
Myrank =  3 Time =     19.733 sec
Myrank =  2 Time =     19.733 sec
```

図 5.4-6 タイマーの前に同期を取る場合

図 5.4-7 にタイマーの前に同期を取らない場合の例を示す．各プロセッサは処理の終了後に他のプロセッサの終了を待たずにタイマーを実行することで，プロセッサ単位の処理時間を正確に計測することが可能になる．

```
    25:                    call MPI_BARRIER(MPI_COMM_WORLD,ierr)
    26:                    t1=MPI_WTIME()
    27:                    sum=0.0d0
    28: +----->            do it=1,m
    29: |+----->            do j=ist,ied
    30: ||V----->            do i=1,j
    31: |||     A              sum = sum + p(it)*a(i,j) - b(i,j)*c(i,j)
    32: ||V-----             enddo
    33: |+-----              enddo
    34: +-----              enddo
    35:                    t2=MPI_WTIME()
    36:                    write(6,6000) myrank,t2-t1
    37:                6000 format("Myrank = ",i2," Time = ",f10.3," sec")

% mpirun -np 4 ./a.out
Myrank =  0 Time =      3.105 sec
Myrank =  1 Time =      8.731 sec
Myrank =  2 Time =     14.291 sec
Myrank =  3 Time =     18.727 sec
```

図 5.4-7 タイマーの前に同期を取らない場合

5.4.2 事例 2(転送回数の削減)

() チューニング方針

図 5.4-8 に MPI データ転送処理の流れを示す. 一般的に MPI の一対一転送では, ①送信側で送信を行うデータをユーザ領域から MPI システム通信バッファ領域にメモリコピーを行い, ②MPI システム通信バッファ間でデータ転送処理を行う. ③受信側で受信したデータを MPI システム通信バッファ領域からユーザデータ領域へメモリコピーを行い, 受信側でデータを参照できる状態にするという手順で行われる. SX-9 の MPI 通信では MPI システム通信バッファ領域を介さない通信である SX-GM(グローバルメモリを利用する通信)を用意しており, メモリコピーを省略することが可能である. しかしながら通信を行う際に同期を取る処理が必要であるだけでなく, 転送処理の立ち上がり時間(レイテンシ)が必要であるため, MPI データ通信処理の回数が増えると実際にデータを転送する時間以外の MPI の処理時間を要するため, 並列化の効果を妨げる大きな要因となる. したがって, 並列化によるオーバーヘッド時間を削減するため, MPI によるデータ通信処理回数の削減を検討する.

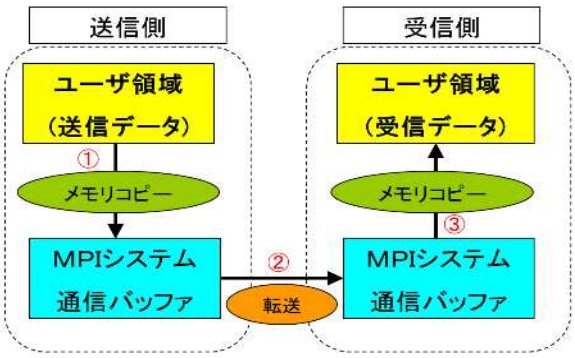


図 5.4-8 MPI データ転送処理の流れ

(2) チューニング内容

図 5.4-9 に通信回数を削減する例を示す. 修正前のリストでは同じ通信相手(ランク)へのデータ転送が 2 回行われている(MPI の通信処理はユーザサブルーチン parallel_sendrecv の中で行われている)が, 修正後のリストでは 2 回分の通信データをまとめる処理を行い(送信データは配列 send_vec に連続して書き込む), 通信処理の回数を 1 回分削減している.

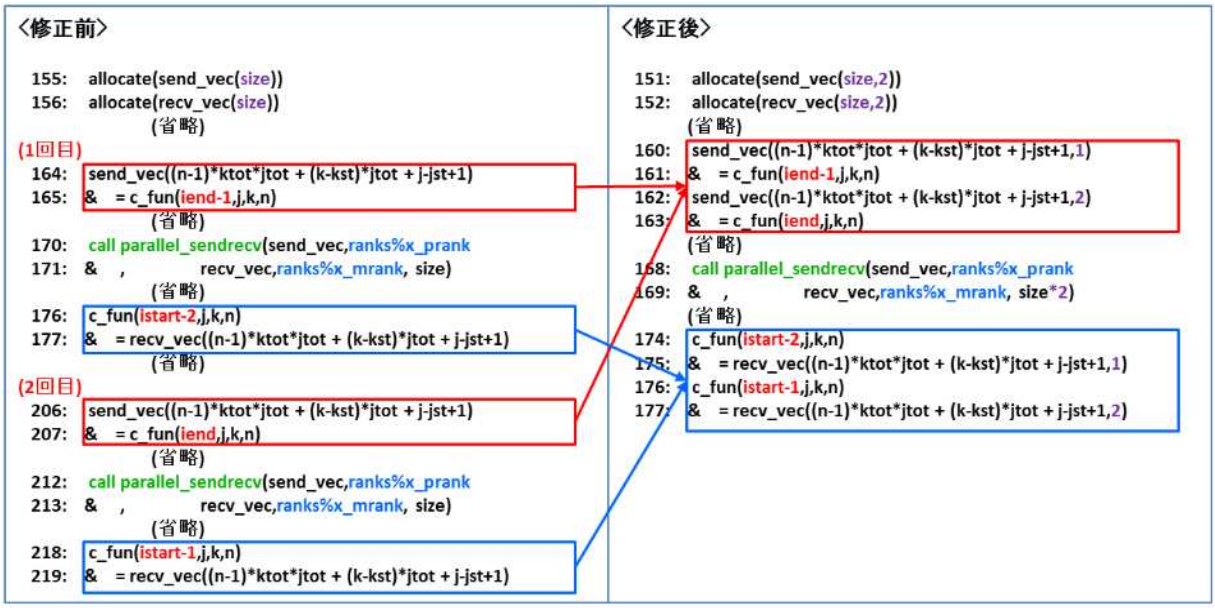


図 5.4-9 通信回数を削減する例

(3) 性能分析

図 5.4-10 に転送回数を削減した場合の FTRACE 情報を示す. 結果は SX-9 の 1 ノードを用いた実験結果である. ユーザサブルーチン parallel_sendrecv の FTRACE 結果で比較を行っている. 呼び出し回数(FREQUENCY)が 1/2 に削減されており, 実行時間は 3.0 秒から 2.4 秒に改善している. データ転送量は同じであるので, 削減されたのは MPI 転送時に発生するオーバーヘッド時間となる.

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	PROC.NAME
55600	3.035	0.055	10826.1	0.5	98.78	252.2	1.852	0.405	0.404	0.133	1.217	【修正前】
27800	2.388	0.086	13436.2	0.3	98.94	253.3	1.686	0.214	0.222	0.074	1.095	【修正後】

図 5.4-10 転送回数を削減した FTRACE 情報

5.4.3. 事例 3(転送量の削減)

(1) チューニング方針

東北大学で開発された BCM(Building-Cube Method)は, 直交格子を用いた計算手法である. 図 5.4-11 に BCM の構成図を示す. BCM の構成は, 計算に使用する物理空間を Cube と呼ばれる様々な大きさの正方体で満たしている. 隣り合うCubeの表面積は4倍, 等倍もしくは 1/4

倍である．さらに各 Cube は同じ数の直行格子で構成されており，この Cube を構成する最小の領域を Cell と呼ぶ．Cube 当たりの Cell 数は等しいことから，Cube 単位の演算量は均等になる性質がある．また Cube ごとに独立して演算を行うことができるが，演算の過程で隣り合う Cube 間でデータの交換が必要になる．

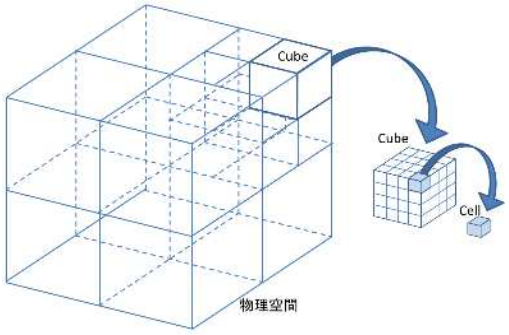


図 5.4- 11 BCM の構成図

MPI を用いる並列化では，Cube 単位で演算量が等しく，演算が独立している特性を利用して，Cube 単位で分割してプロセッサに割り当てる．そのため隣接 Cube を同じプロセッサが担当する場合は，MPI データ転送処理の必要はないが，異なるプロセッサが隣接 Cube を担当する場合は MPI データ転送処理により情報交換に必要なデータをアクセス可能な領域に転送する必要がある．図 5.4- 12 に隣接 Cube 間のデータ転送のイメージを示す．BCM のモデルによって情報交換に用いる Cell 数(境界要素数)は異なるが，一般に Cube に内包する Cell 数が少ない場合は，大部分の Cell が Cube 間の情報交換に必要になり，Cube 全体を転送する方が処理時間を短縮できると考えられる．しかし Cube に内包する Cell 数が多い場合は，Cube 全体を転送すると隣接 Cube 間の情報交換に必要のないデータも一緒に転送することになり，転送量は膨大になる．そのような場合は，隣接 Cube 間の情報交換に必要な Cell のみを転送することで，転送量の削減を図る必要がある．しかしながら，Cube 内の Cell は面の方向によっては連続したメモリ空間に格納されておらず，データを連続になるように詰替えてから MPI データ転送する必要がある．また転送後は元の格納形式に詰め戻す処理が必要になる．したがって，必要なデータのみを MPI データ転送する手法は，Cube 内の Cell 数がある程度以上内包されていて，データの詰替えや詰め戻し処理にかかる時間を加味しても全体の処理時間を短縮することが可能な場合にのみ適用することになる．

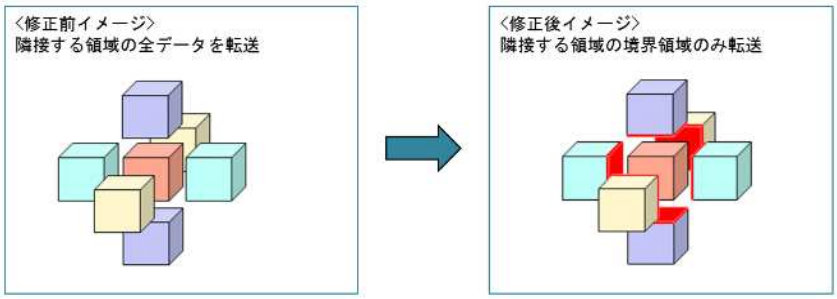


図 5.4- 12 隣接 Cube 間のデータ転送

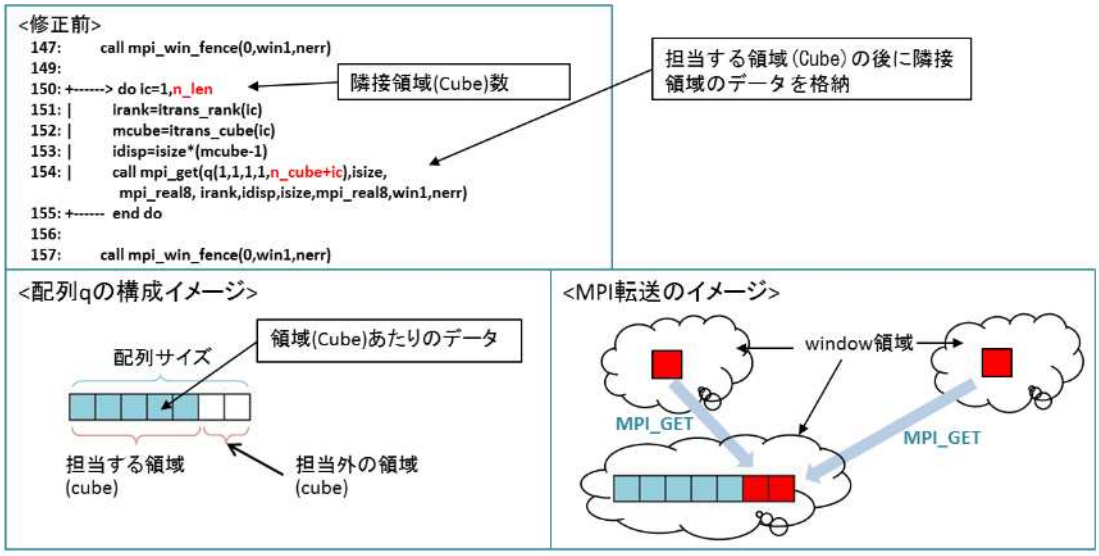


図 5.4-13 修正前(Cube 全体を転送)の処理

(2) チューニング内容

本プログラムでは，図 5.4-13が示すように，隣接するCube間の情報交換のためのMPIデータ転送処理に単方向通信MPI_Getを使用している(単方向通信はMPI-2の仕様に属する転送方法)．予め定義したウィンドウ領域上にあるデータを MPI_Win_Fence で囲まれた間はその領域を担当するプロセッサの動作に関係なく自由に参照または更新を行うことができる(MPI_Win_Fence 間はウィンドウ領域内のデータの内容が MPI データ通信により参照または更新の可能性があるので，プログラムの処理側で参照または更新することはできない)．配列 q はプロセッサが担当する領域だけでなく，MPIデータ転送する隣接 Cube の情報を格納する領域を確保しておき，Cube 間の情報交換に必要なデータは MPI_Get を用いてその情報を担当するプロセッサのウィンドウ領域から所得する．この時MPI_Getで取得するデータ量はCube全体の大きさになる．図 5.4-14 に修正後(Cube 面を転送)の処理のイメージを示す．転送処理の前に Cube 間の情報交換に必要なデータをメモリ上で連続領域となるように作業配列に詰める処理(ユーザサブルーチン mpi_packing)が必要となる．また転送処理の後は作業配列を元の配列の形式に詰め戻す処理(ユーザサブルーチン mpi_unpacking)が必要になる．

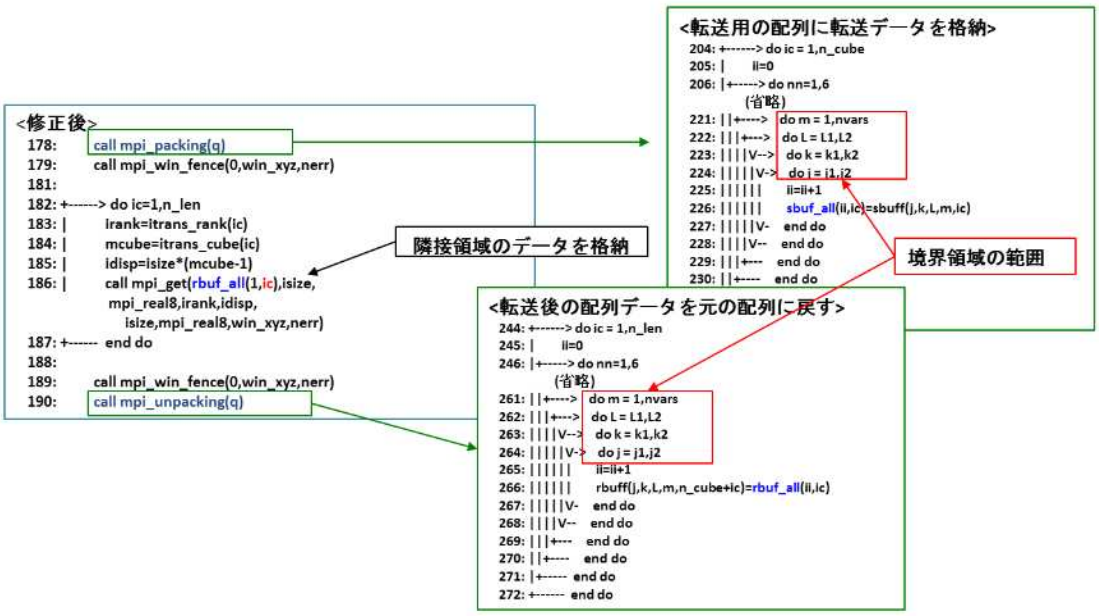


図 5.4-14 修正後(Cube 面を転送)の処理

(3) 性能分析

図 5.4-15 に転送量削減の効果を検証した FTRACE 情報を示す. 実験は SX-9 の 1 ノードを使用している. Cube 内の Cell 数は 4,096 個である. Cube 全体を MPI データ転送していた修正前では一回当たりの転送量は 13.1MByte であったが, Cube 間の情報交換に必要な Cell のデータのみ MPI データ転送するように修正することにより, 一回当たりの転送量を 1.1MByte に削減することが可能になった.

ELAPSED TIME[sec]	COMM.TIME [sec]	COMM.TIME / ELAPSED	IDLE.TIME [sec]	IDLE.TIME / ELAPSED	AVER.LEN [byte]	COUNT	TOTAL.LEN [byte]	PROC.NAME
14.178	14.037	0.990	0.001	0.000	13.1M	111500	1.4T	【修正前】
1.462	1.263	0.870	0.001	0.001	1.1M	111500	122.1G	【修正後】

図 5.4-15 転送量削減の効果を検証した FTRACE 情報

高速化推進研究活動報告　第６号

2015年２月発行

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