高速化推進研究活動報告

第5号

2011年9月

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５号

目 次

1 高速化推進研究活動報告の刊行にあたって

東北大学理事(教育・情報システム担当)
根元 義章

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、約 40 年が経過した。本センターは、わが国の計算機の揺籃期に、期待をもって誕生し、時代とともに進展をとげ、計算科学・計算機科学に関する共同利用・共同研究を通して学術研究の発展に大きく貢献してきた。これも文部科学省をはじめ、関係各位のご支援、ご協力の賜物であり、厚く感謝申し上げる。

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」ということに尽きる。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きた。その後も現在にいたるまで、種々の情報技術やネットワーク技術の進展にあわせて、上記 3 者の協力による情報技術の開発が進められてきている。特に、1997 年 9 月より、利用者、本センター、日本電気㈱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た。本共同研究の取り組みは、その成果として単にその利用者のプログラムの高速化だけにとどまらず、得られた高速化技術を広く一般利用者に還元させることにより、スーパーコンピュータの潜在能力を最大限に引き出すプログラム開発に大いに貢献している。さらに、次期スーパーコンピュータシステムの研究開発に取り組むセンター教職員、ベンダー技術者双方にとっては、利用者の最先端のシミュレーションプログラムはシステム設計をする上で重要な評価指標を与えてくれるものである。「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 5 号は、2008 年度から 2010 年度までに得られた成果が収録されている。本共同研究は、本センターの技術系の職員を中心とするスタッフ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。常に最新最先端のシステムの運用とその高度利用技術を利用者に提供し続けるという本センターの使命を果たすためには、今後とも、本研究活動を精力的、かつ継続的に取り組んでいくことが肝要であると考えている。

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、日本電気㈱側のスタッフならびに、本センター側スタッフに厚くお礼を申し上げる。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ある。

2 高速化推進研究活動報告

サイバーサイエンスセンター　センター長
スーパーコンピューティング研究部　教授
小林広明

2.1 はじめに

近年，スーパーコンピュータを用いた科学技術シミュレーションは，理論，実験に次ぐ第三の科学として，先端科学や工学分野において重要な役割を担っている．東北大学サイバーサイエンスセンター(以下，本センター)は，その前身である大型計算機センターの設立(1969 年)以来，図 2.1 に示す様に，研究室のレベルを遙かに超える最高性能の計算機システムを設置し，最先端の学術研究を強力に支援・推進してきた．さらに，利用者にとって使い勝手の良いシステムの構築，他では実行できない大規模プログラムの実行環境の整備を行うと共に，スーパーコンピュータの性能を最大限に発揮することを目的に，利用者，本センター教員・技術職員，日本電気(株)(以下 NEC)が一体となって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の研究・開発に取り組み，計算科学・計算機科学の発展に貢献してきた．本章では，はじめに本センターが有する大規模科学計算システムの概要について述べ，次に本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，およびスーパーコンピューティング研究部と NEC の共同研究活動について説明する．

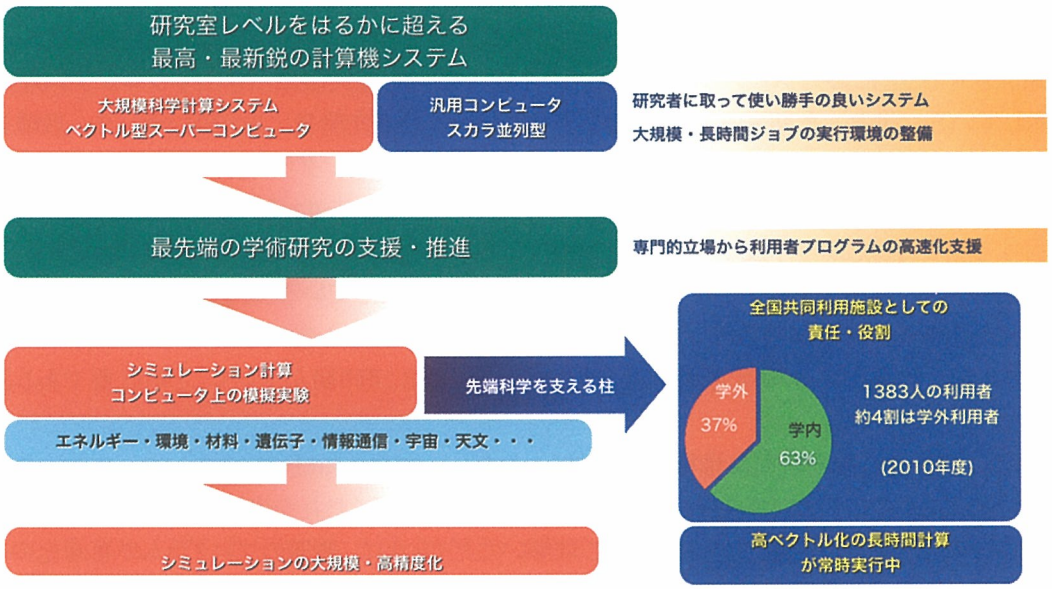

図 2.1　サイバーサイエンスセンターの役割

2.2 大規模科学計算システム

平成22年度4月，本センターの有する大規模科学計算システムの更なる計算性能の向上とユーザの多様なニーズに応えることを目的に，システムの設計・更新を行った．本システム更新では，計算サーバであるベクトル型スーパーコンピュータSX-7Cとフロントエンドサーバであるtx7/i9610をリプレースすることで，ベクトル性能ばかりでなく，スカラ性能の大幅な性能向上を図っている．図2.2に本章で取り上げる本センターの新しい大規模計算システムの構成を示す．

旧システムにおけるベクトル型スーパーコンピュータSX-7C（ピーク性能640Gflop/s, 40GB）は，2 ノードのSX-9(3.2Tflop/s, 2TB）にリプレースされ，既に導入済みの16ノードのSX-9 と併せてその理論性能は29.4Tflop/s となり，全国の本センターユーザのベクトル型計算機に対する高いニーズに応えられるシステム構成としている． また，TX7/i9610は最新のIntel Xeon X7560プロセッサを搭載するExpress5800/A1080a-Dにリプレースされた．Express5800/A1080a-Dは，ノードあたり4つのCPU(32コア)，512GBのメモリを搭載し，今回導入した3ノードの理論性能は1.74Tflop/sに達する．これらのリプレースにより，大規模科学計算システムは，ベクトル性能とスカラ性能の大幅な性能向上が実現できている．

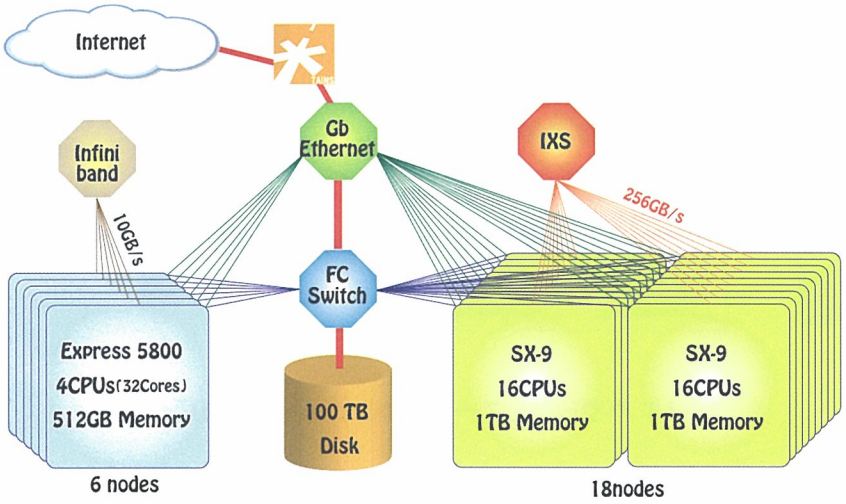

図2.2 新大規模科学計算システム

本センターでは，1986年よりベクトル型のスーパーコンピュータの運用しており，平成20年3月には，世界に先駆けベクトル型スーパーコンピュータSX-9を導入した．高いメモリバンド幅を有し，科学技術シミュレーションを高い実効効率で実行可能なベクトル型スーパーコンピュータは，図2.3に示す様々な分野で重要な役割を担っている．また，本センターでは，2004年より世界的な評価指標として近年注目を集めているHPCCベンチマークによるスーパーコンピュータの性能評価に国内でいち早く取り組んでいる．HPCCベンチマークでは，これまでスーパーコンピュータの性能評価で重要視されてきた総演算性能の評価に加えて，アプリケーション実行におけるスーパーコンピュータの実効性能を引き出す上で重要なメモリアクセス性能とネットワーク性能の評価と，多くのアプリケーションで頻繁に使用されるカーネルコードを用いたシステムの総合性能評価が可能になっている．これにより，従来のノード数に頼るスーパーコンピュータの数量的な評価ばかりでなく，スーパーコンピュータの「質」と「有用性」を評価することが可能となる． 2008年11月に行ったHPCCチャレンジベンチマークを用いた評価では，本センターで運用するベクトル型スーパーコンピュータSX-9がCray, IBM, SGIなどのスカラ型スーパーコンピュータとの比較において，28の評価項目のうち19項目でその当時(2008年11月)世界最高性能値を達成した(図2.4)．特に，HPCチャレンジベンチマークの主要ベンチマークであるGlobal FFTにおいては，ランキングは第4位ながらも，他の世界有数のシステムとの比較において，SX-9の極めて高い実効効率を実現可能であることを示すことで，ベクトル型スーパーコンピュータの科学技術アプリケーションにおける高い有用性を明らかにした(図2.5)．

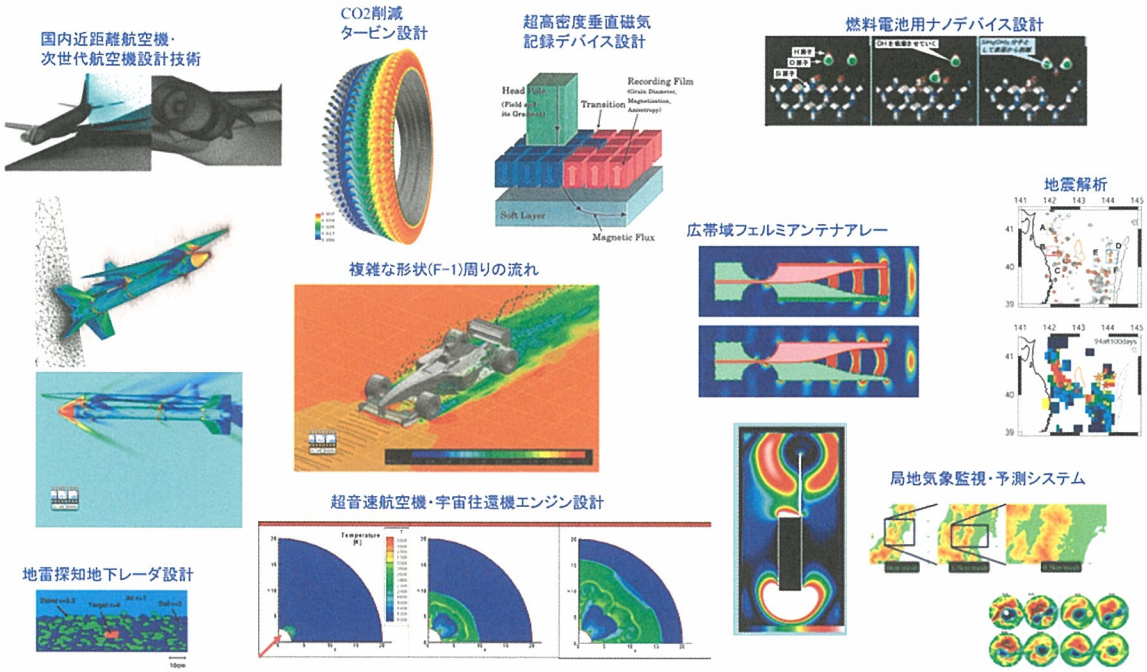

図 2.3　ベクトル型スーパーコンピュータの利用分野

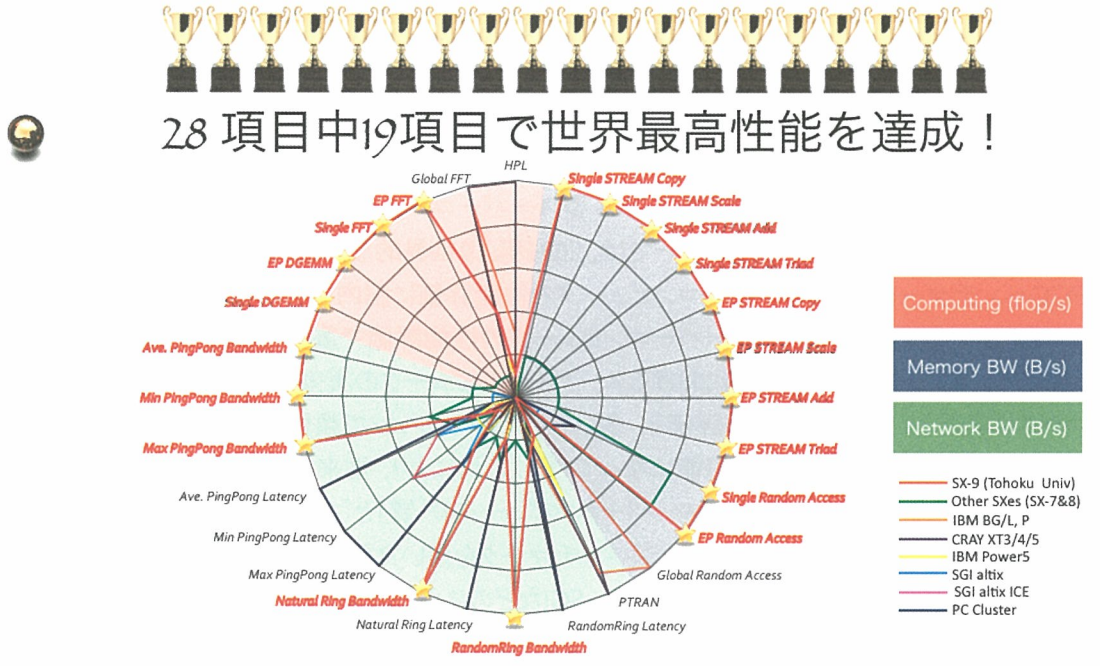

図 2.4 HPC チャレンジベンチマーク（総合性能）

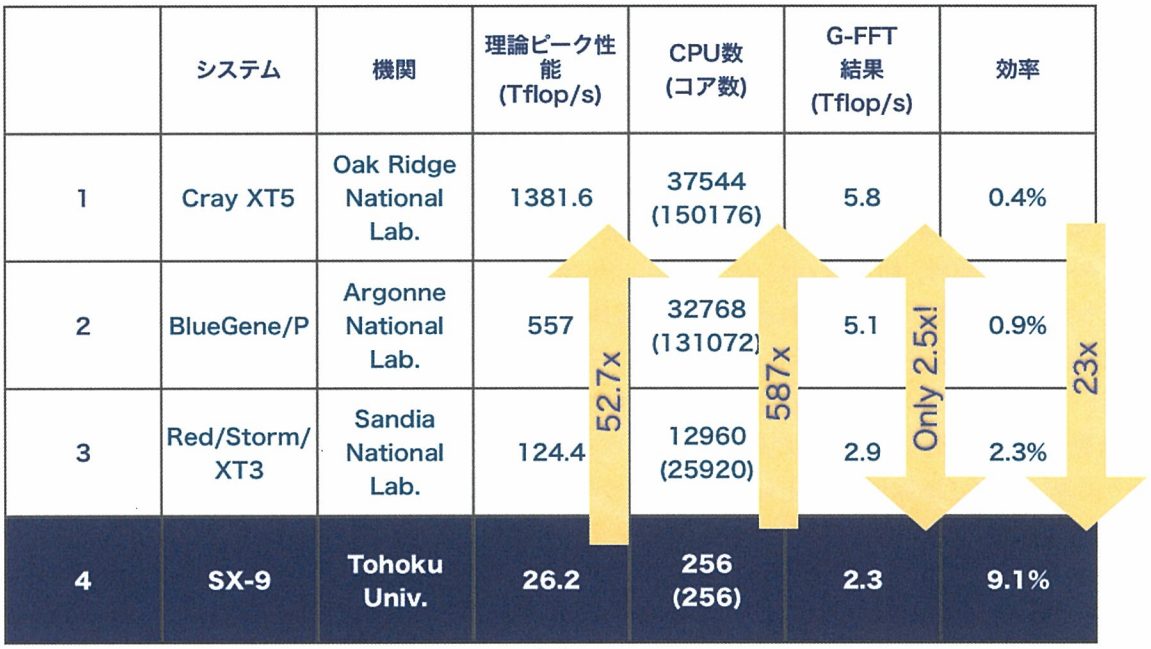

図2.5 HPCチャレンジベンチマーク(G-FFT)

高いベクトル化率が達成困難な大規模並列アプリケーションの高効率実行と，Gaussian，Marc，Marc Mentat，Mathematica，MATLAB，SAS 等の汎用アプリケーションの高速実行環境の提供を目的に，2010年4月にExpress5800/A1080a-Dを3ノード導入し，スカラ性能の向上を図っている．旧並列コンピュータと新たに導入したExpress5800/A1080a-Dの性能比較を表2.1に示す． 各ノードは，Infinibandを介して高速に相互接続されており，ノードを跨いで実行される高い並列性を有するアプリケーションの高速実行も可能にしている．

表2.1 新並列コンピュータ諸元

		TX7/i9610	Express5800/ A1080a-D	向上比
コア	動作周波数	1.6GHz	2.26GHz	1.4
	最大演算能力	6.4Gflops	9.06Gflops	1.4
CPU ソケット	コア数	2	8	4.0
	最大演算能力	12.8Gflops	72.4Gflops	5.6
	メモリバンド幅	8.3GB/s	34.1GB/s	4.1
	L3 キャッシュ容量	24MB	24MB	1.0
SMP ノード	最大演算能力	409.6Gflops	289.9Gflops	0.7
	メモリ容量	512GB	512GB	1.0
システム	ノード間通信速度	800MB/s	4GB/s	5.0
	最大演算能力	1.2Tflops	1.74Tflops	1.4
	メモリ容量	1.5TB	3TB	2.0

2.3 高速化推進研究活動および共同研究

コンピュータの機能・性能は，半導体加工技術，コンピュータアーキテクチャ，ソフトウェア技術の進歩に伴い，著しい速度で向上しており，本センターの大規模科学計算システムもシステムのリプレースのたびに，前システムの約10倍の計算能力をユーザに提供している．

しかし，その潜在能力を最大限に享受し，大規模且つ高精度な計算機シミュレーションを実現するためには，コンピュータシステム，およびプログラミング技術に関する高度な知識が求められるのが実情である．このような状況下で，本センターでは「最先端，かつ世界最高性能クラスのコンピュータシステムの導入，利用環境の整備・運用・研究・開発」という使命を果たすために，1997年より全国の情報基盤センター（7センター）に先駆け，利用者・センター運用スタッフ（技術系職員・スーパーコンピューティング研究部教員），NECの技術者から構成される高速化支援体制を整備している(図2.6)．この支援体制のもと，自動ベクトル化，自動並列化機能のみに頼ることなく，実シミュレーションコード解析に基づく臨床学的研究開発を行い，ユーザプログラムの高速化支援活動に取り組んできた． 表2.2に，1999年から2009年にかけて本活動を通して，取り組んできた高速化支援の結果を示す．この間，合計142件のプログラムの高速化に取り組み，平均で単体性能向上比は10倍，並列性能向上比8.5倍を実現しており，高速化支援体制による支援が高い成果を上げていることがわかる．

さらに，1999年からは全国に広がる本センターのユーザ(計算科学者)とセンター(計算機科学者)が連携し，サイバーサイエンスセンター共同研究として，106件の共同研究を推進している．また，2008年からは文部科学省先端研究施設共用促進事業，2010年からは大規模情報基盤共同利用・共同研究拠点の一拠点として，全国の計算科学者らとの共同研究も推進している．特に，文部科学省先端研究施設共用促進事業を通して，本センターのベクトル型スーパーコンピュータSX-9を用いた民間企業・大学研究者との共同研究を強力に推進し，初の国産短距離ジェット開発，次世代ハードディスクの要素技術開発や，高効率蒸気タービン開発に大きく貢献している．

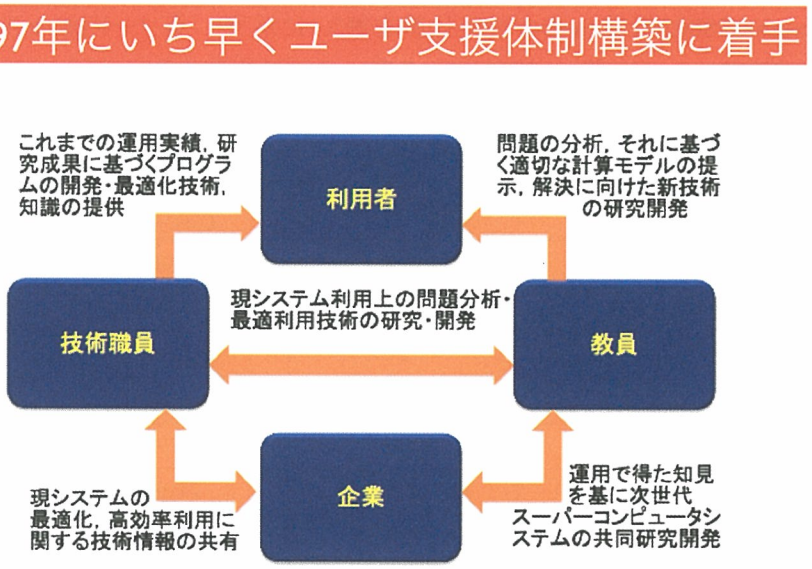

図2.6 高速化支援体制

表2.2 高速化支援活動成果

年	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009
件数	8	9	10	7	18	20	8	29	10	15	8
単体性能向上比	4.5	2.5	1.6	2.2	6.7	2.9	1.5	3.05	33	9.3	47
並列性能向上比	31.7	8.6	4.9	2.8	18.6	4.5	4.1	8.0	1.9	5.05	3.6

※単体・並列性能向上比：オリジナルの性能を1としたときの性能向上

これらの一連の高速化支援・ユーザとの共同研究活動を通して，本センターのスーパーコンピューティング研究部は，現有スーパーコンピュータの高度利用技術の研究開発ばかりでなく，NECと共同して次期スーパーコンピュータの要素技術の研究開発に取り組んでいる．

現有するシステムの高度利用技術に関する成果としては，SX-9に新たに追加されたADB(Assignable Data Buffer)を有効に利用することで、大幅な性能向上が得られることを明らかにしている．この成果は，高性能計算に関するトップクラスの会議であるSC09の一般講演（2009年は日本からの採択2件）として発表し，大きな注目を集めた．

また，次期スーパーコンピュータの要素技術の研究開発としては，3次元積層技術を用いたチップマルチベクトルプロセッサのアーキテクチャ設計や，将来の大規模なオンチップメモリを搭載するベクトル型スーパーコンピュータのためのソフトウェアチューニング戦略に関する研究を推進した(図2.7)．これらの研究成果は，学術論文誌や，SCなど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，2011年にはHPCS(ハイパフォーマンスコンピューティングと計算科学シンポジウム)において，IEEE Computer Society Japan Chapter 優秀若手研究賞を受賞している．
(http://www.is.tohoku.ac.jp/detail/kobayashi110124.html)

この他にも，2006年から毎年秋にドイツ・シュッツットガルト大学高性能計算センター(HLRS)と共同で，高性能計算に関する国際ワークショップTeraflop Workshopの開催や，SCに置けるブース展示，2010年にはFIT2010においてベクトルコンピュータに関するパネルセッションの企画など，成果の国内・国外への発信を精力的に行っている(図2.8，図2.9)．

これら，国内外で高く評価されている活動・成果はいずれも，利用者・本センターの教員・技術職員・NEC の技術者が密に連携した，高速化支援体制・共同研究体制が基になっている．高速化支援研究のためには，研究目的や研究内容は勿論，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，ユーザとのミーティングを重ね，本センターと NEC のスタッフが長い時間をかけてこれらを理解し，本センターの大規模科学計算システムのアーキテクチャを考慮したアルゴリズム，プログラミング，データ構造についてユーザに提案してきた．

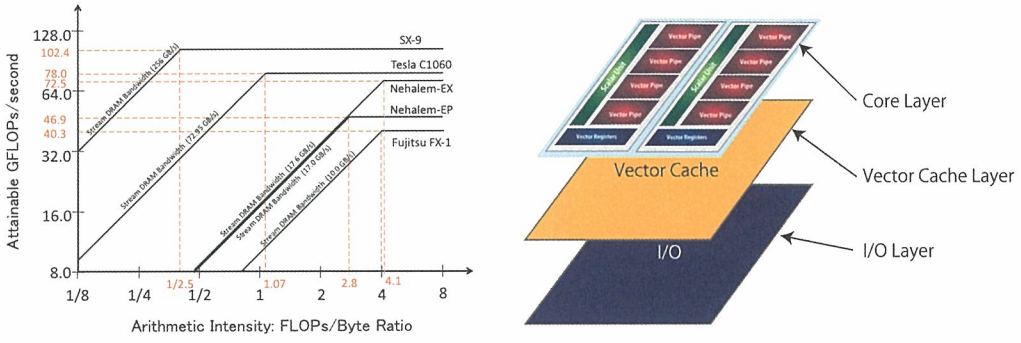

図 2.7 ルーフラインモデルを用いたプログラムチューニングと 3DCMVP アーキテクチャ

図2.8 SCおける発表

図2.9 Teraflop workshop, FIT2009パネル

2.4 まとめ

本章では，本センターの有する大規模科学計算システムと，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 2008 年 11 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の推進には，利用者の協力を欠かすことができない．本報告書記載されている成果は，東北大学長谷川昭名誉教授，澤谷邦男教授，中橋和博教授，小野高幸教授，山本悟教授，岩崎俊樹教授，佐々木大輔助教，笹尾泰洋助教，東京大学　押山淳教授，東京理科大学　塚原隆裕教授，大阪大学　森川良忠教授，物質・材料研究機構　岩長祐伸博士，海洋研究開発機構　有吉慶介博士，東京農工大学　高橋俊助教らと，本センターの共同研究を推進した先生方の多大なるご協力により得られたものである．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
伊藤英一、大泉健治、小野敏、山下毅

本センターのスーパーコンピュータシステムは、2008年4月にスーパーコンピュータSX-9に変更され、それまでのSX-7システムに比べてベクトル性能が12.5倍、メモリ容量が8倍に拡大した。SX-9システムは、2008年4月にサービスを開始してから2011年3月で3年を経過した。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して、さらに長時間ジョブは実行時間の推定が困難であるため CPU 時間を制限しないような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極限まで高めておくことが重要である。そのためにはスーパーコンピュータのハードウェアやコンパイラについての深い知識が要求される。また、コンパイラの自動ベクトル化、自動並列化の機能強化には、利用者プログラムに積極的に係わっていく必要もある。そこで、本センターでは利用者のプログラムを高速に処理させるための活動に専門の担当者を置き、プログラムチューニングなどの高速化支援を行っている。

以下では、2008年度から2010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(2008年～2010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2008年度(平成20年度)、表3.1.2に2009年度(平成21年度)、表3.1.3に2010年度(平成22年度)、の高速化支援性能向上比と概略を報告する。

表3.1.1 2008年度 (20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(8並列)
1	4倍精度複素数を倍精度に変更	16倍	—
2	指示行によるループ長拡大	1.2倍	—
3	バンクコンフリクトの低減	1.1倍	—
4	インライン展開 多重ループの融合	70倍	5.3倍 (8並列)
5	インライン展開 無駄な処理の削減 ショートループのベクトル化抑止	2.1倍	—
6	ファイル出力の削減	1.7倍	—
7	リストベクトルのベクトル化	1.2倍	—

8	ループ構造の修正	5.0 倍	—
9	並列処理時の待ち合わせ方式変更	—	2.8 倍 (8 並列)
10	ベクトルレジスタの活用 スカラ処理の削減 ベクトル長の拡大 平面法による並列化	3.2 倍	12.8 倍 (16 並列)
11	リストベクトルの削減によるメモリ負荷の軽減 多重ループの融合 ADB の利用	3.0 倍	2.5 倍 (16 並列)
12	ベクトル化促進	1.2 倍	4.4 倍 (16 並列)
13	インライン展開 グローバルメモリ使用による通信時間の短縮 ADB の利用	—	2.5 倍 (64 並列)
14	メモリアクセスの削減	1.3 倍	—
15	ファイルアクセス頻度の抑制 高速ディスク装置の利用	15 倍	—

表 3.1.2 2009 年度 (21 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 (8 並列)
1	インライン展開と outerunroll 指示行/ADB 指示行に よる Memory Network Conf 削減		5.9 倍
2	IF 文の移動と除去による演算数削減とバンクコンフリクト削減 2 重ループの 1 重化によるベクトル長の拡大	381 倍	
3	データ依存関係の解消による並列化促進		1.4 倍
4	並列版 ASL の提供		1.5 倍
5	MPI 通信部分の改善		2.6 倍
6	FFT 部分の改善		2.2 倍
7	ソースコード修正による MPI 並列化促進		1.6 倍
8	ソースコード修正による並列化促進		10 倍

表 3.1.3 2010 年度 (22 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 (8 並列)
1	配列サイズの奇数化によるバンクコンフリクト削減		32.6 倍

	MathKeisan ライブラリの提供		
2	IF 文の移動と除去による演算数削減とバンクコンフリクト削減 作業配列の導入によるベクトル化率の促進	18.5 倍	3.8 倍 (4 並列)
3	ループの入れ換えによるベクトル化率の促進 配列サイズの奇数化によるバンクコンフリクト削減	2.0 倍	
4	平面法によるベクトル化率の促進	25.8 倍	109 倍
5	IF 文の除去によるベクトル化率の促進	3.1 倍	
6	ASL ライブラリによるベクトル化率の促進	278 倍	
7	作業配列の導入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	
8	ループ融合の防止によるバンクコンフリクト削減	1.7 倍	

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 2008 年度から 2010 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 に示す。表は 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 時間に対して千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%	28	11	11	32	32	32	44	56	75	586
80%	1	0	1	0	1	1	1	0	1	23
70%	2	1	0	0	0	0	1	1	0	2
60%	1	1	1	1	0	0	0	0	0	4
50%	0	0	0	0	0	0	0	0	0	2
40%	1	1	0	0	0	0	0	0	0	1
30%	0	0	0	0	0	0	0	0	0	2
20%	0	0	0	0	0	0	0	0	0	1
10%	0	0	0	0	0	0	0	0	0	2
0%	16	10	3	0	0	0	0	0	0	6

並列化率

備考：マトリックスの値は、ジョブ CPU 時間の千分率（全 CPU 時間比）

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 59%
・ベクトル化率が 90%以上のジョブ	約 91%
・並列化率が 80%以上のジョブ	約 71%

このように、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 3.2.2 は 2008 年度から 2010 年度までの 3 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。

表 3.2.2 CPU 時間使用分布

時間	2008 年度	2009 年度	2010 年度
0 ～ 99	18%	16%	8%
100 ～ 999	49%	38%	39%
1000 ～1999	13%	12%	13%
2000～	20%	34%	40%

3.3 今後の取り組み

SX-9 システムの運用を開始してから 3 年を経過し、ジョブの大規模化、長時間化が一層進んでいる。3.2 節に述べたようにベクトル化率、並列化率とも高いところに集中している。並列化率 80%未満のジョブの割合が 29%あり、今後は並列処理のチューニングに力を入れていく必要があると考えている。

4　スーパーコンピュータ SX-9 の高速化

スーパーコンピューティング研究部　江川隆輔　岡部公起
情報部情報基盤課　伊藤英一　小野敏　山下毅
日本電気株式会社　撫佐昭裕　神山典　小久保達信　吉村健二
遠藤清隆　小沢実希　坂本英顕　金野浩伸　坂口祐太
NEC システムテクノロジー株式会社　曽我隆

4.1 SX-9 の特徴

SX-9 は高速ベクトルプロセッサと大規模主記憶装置を有するベクトル型スーパーコンピュータである. 東北大学サイバーサイエンスセンターでは 18 ノードの SX-9 を導入しており, そのうち 16 ノードがノード間接続装置(以下, IXS)に接続されたマルチノードシステムである. 表 4.1-1 に SX-9 の主要諸元を, 表 4.1-2 に SX-9 マルチノードシステムの主要諸元を示す.

表 4.1-1　SX-9 主要諸元

項目			諸元
最大ベクトル演算性能			1.6TFLOPS
CPU 数			16
CPU	レジスタ	ベクトルレジスタ	144KB
		ベクトルマスクレジスタ	256bit×16
		スカラレジスタ	64bit×128
	データ形式	固定小数点	32/64bit
		浮動小数点	32/64/128bit
		*128bit はスカラ命令のみサポート	IEEE
		論理	64bit
	ベクトル演算パイプライン		5 種類×8 セット
	ADB		256KB
	スカラ演算パイプライン		1 セット
	キャッシュ		命令:32KB
			オペランド:32KB
主記憶装置	容量		1TB
	最大データ転送能力		4TB/s

表 4.1.2　SX-9 マルチノードシステム主要諸元

項目	諸元
ノード数	16
最大ベクトル演算性能	26.2TFLOPS
CPU 数	256
主記憶装置容量	16TB
IXS 転送性能(片方向)	128GB/s (ノード当たり) 2TB/s (システム全体)

(1)　CPU の特徴

SX-9 に搭載されている CPU は, 単一チップで 102.4GFLOPS のベクトル演算性能を有するベクトル

プロセッサである. 図 4.1-1 に SX-9 のプロセッサ構成を示す. CPU 内部はベクトルユニットとスカラユニットに分かれており, ベクトルユニットは 8 つのベクトルパイプラインセットで構成されている. 各ベクトルパイプラインセットはマスク演算器, 論理演算器, 乗算器×2, 加算器×2, 除算器/平方根演算器を有している. また, SX-9 から MAX/MIN 関数がハードウェア命令化され, 最大値・最小値の計算時間が短縮されている.

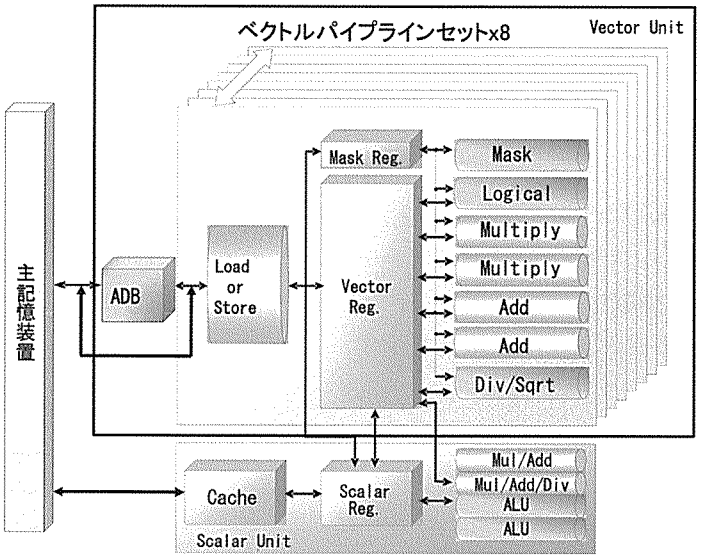

図 4.1-1 SX-9 プロセッサの構成図

(2) メモリシステムの特徴

SX-9 は, CPU と主記憶装置(MMU)間の距離が論理的に等しい共有メモリ型計算機である. 図 4.1-2 に SX-9 のメモリ構成を示す. SX-9 は CPU と主記憶装置間にネットワークルータ(以下, RTR)という内部スイッチング機構を配することで, 広いメモリバンド幅と大規模共有メモリを実現している. また, 主記憶装置はノードあたり 32,768 個のメモリバンクで構成されている. バンクのアクセスに関しては SX-7C までの SX シリーズでは一度に 1 演算要素(8Byte)を処理していたが, SX-9 から 2 演算要素(16Byte)を処理するように強化し, ロード・ストアの処理時間の短縮を図っている.

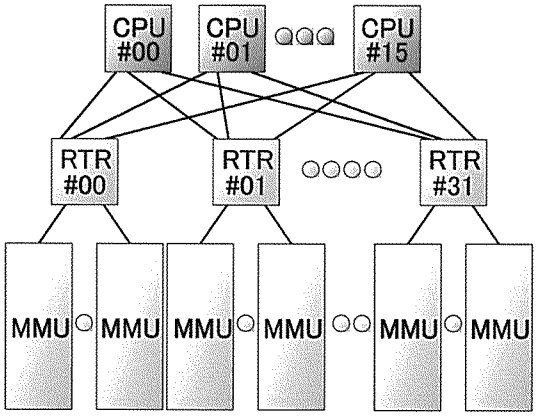

図 4.1-2 SX-9 のメモリ構成図

SX-9 では CPU－主記憶装置間に ADB(Assignable 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構を有している. 図 4.1-3 に ADB の概念図を示す. ADB は CPU－主記憶装置間よりメモリバンド幅が広く, メモリレイテンシ(遅延)が短いという特徴を持っている. 表 4.1-3 に SX-9 の主記憶装置と ADB のデータ供給性能を示す. データ供給性能は, 演算性能あたりのデータ供給量を表す指標としてベクトル演算性能あたりのメモリバンド幅(Byte/FLOP)である.

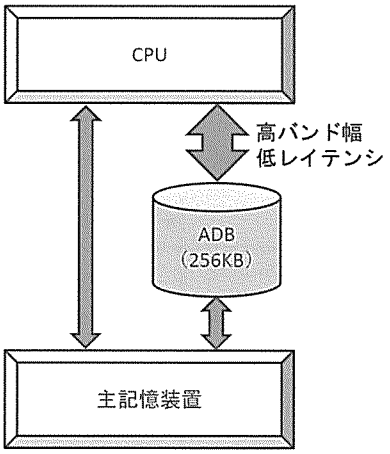

図 4.1-3 ADB の概念図

表 4.1-3 SX-9 の主記憶装置と ADB のデータ供給性能

	CPU-主記憶装置間	CPU-ADB 間
バンド幅	256.0GB/s	409.6GB/s
ベクトル演算性能あたりのバンド幅	2.5Byte/FLOP	4.0Byte/FLOP

(3) ソフトウェアの特徴

① コンパイラ

SX-9 用コンパイラとして FORTRAN90/SX と C++/SX が利用可能である. FORTRAN90/SX は ISO/IEC 1539-1:1997, ISO/IEC1539:1991 に準拠している. C++/SX は ISO/IEC 9899:1999, ISO/IEC 14882:1998 に準拠している.

FORTRAN90/SX, C++/SX は自動ベクトル化機能, 自動並列化機能を有しており, ユーザは意識することなく, ベクトル化, 並列化することができる.

② MPI ライブラリ

SX-9 用 MPI ライブラリは MPI-1.2 および MPI-2 に準拠した機能を提供している.

③ 科学技術計算ライブラリ

SX-9 用ライブラリとして, 科学技術計算ライブラリ ASL, 数値計算ライブラリ MathKeisan が利用可能である.

4.2 ベクトル処理による高速化

(1) ベクトル処理の概要

科学技術計算プログラムの多くは特定の DO ループに実行の大部分が集中し, DO ループ内の配

列データ(ベクトル)に対し繰り返し演算が行われている. この規則性に着目して高速化を図った方式がベクトル処理である. ベクトル処理はDOループによる配列データへの繰り返し演算を, ベクトルユニットを用いて一括に実行するものである. 一方, スカラ処理は繰り返し演算を逐次的に実行するものである.

(2) 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 4.2-1 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.

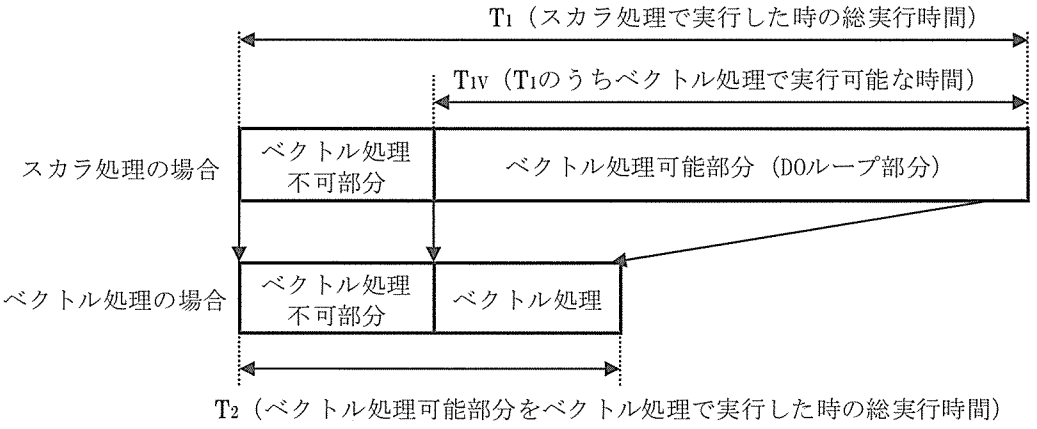

図 4.2-1 ベクトル処理による実行時間短縮のイメージ

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 4.2-2 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.

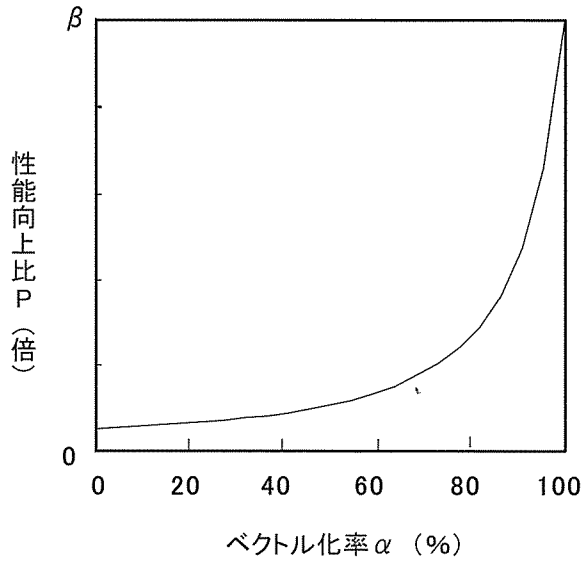

図 4.2-2 ベクトル化率と性能向上比（アムダールの法則）

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．後述の PROGINF や 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る．ベクトル演算率は，プログラムで処理される全演算要素数に対するベクトル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．

(3) 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 4.2-3 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．図 4.2-4 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．

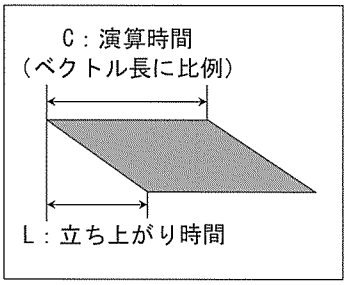

図 4.2-3 立ち上がり時間

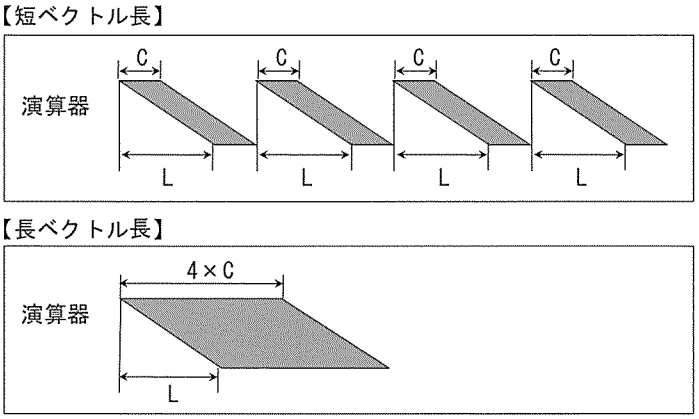

図 4.2-4 ベクトル長と立ち上がり時間

(4) メモリアクセス

ベクトルユニットはベクトル処理によって複数の演算要素を一括で処理することが可能である. この処理を高速に行うためには, 主記憶装置からベクトルユニットに対して, データを滞りなく供給することが必要である.

データ供給の遅延の要因にバンクコンフリクトがある. バンクコンフリクトは CPU ポート競合とメモリネットワーク競合の 2 つに分類される. CPU ポート競合は CPU 内における同一のポートにロード・ストアが集中した時に発生する競合を指す. メモリネットワーク競合は同一のメモリバンクへのアクセスで発生する競合や CPU と主記憶装置間の経路上で発生する競合を指す. バンクコンフリクトが発生するとデータ供給能力が低下し, 高速化の妨げの要因になる. バンクコンフリクトを回避するためには, メモリアクセスの間隔が偶数となることを避け, 可能ならば連続アクセスとすることや, ループのアンロールを行うなどしてメモリアクセス回数を減らすことが重要である.

また, メモリアクセスの効率化では, 前述の ADB の活用があげられる. ADB を利用することで主記憶装置からのデータ供給性能を補うことや, メモリレイテンシを短くすることが可能となり性能向上を図れる場合がある. 後述のチューニング例 4.3.8(4)を参考に ADB の利用を試みて頂きたい.

4.3 高速化技法

4.3.1 多重 DO ループの一重化による高速化

(1) 多重 DO ループの一重化の概要

多重 DO ループの一重化とは, 図 4.3-1 のような多重構造の DO ループを, 図 4.3-2 のように一重の DO ループに書き換えることをいう. 図 4.3-1 では, 内側の DO ループはベクトル長 10 でベクトル処理され, 外側の DO ループはこのベクトル処理が 20 回実行されるので, ベクトル処理の立ち上がり時間は 20 回分必要となる(4.2.(3) 図 4.2-4 参照). 一方, 図 4.3-2 では, ベクトル長 200 のベクトル処理を1回だけ実行すればよいので, 立ち上がり時間は1回分だけですみ, 高速化することができる. ここで, 配列 a(ij,1), b(ij,1), c(ij,1)は一重化された DO ループで, 元の配列 a(i,j), b(i,j),c(i,j)を連続してアクセスするために書きかえたものである.

多重 DO ループの一重化は, ここで示した例のように, 内側の DO ループの長さが短いときに特に効果がある.

```
!多重 DO ループ
real,dimension(10,20)::a,b,c
do j=1,20
  do i=1,10
    a(i,j)=b(i,j)+c(i,j)
  enddo
enddo
```

図 4.3-1 多重 DO ループの例

```
!図 4.3-1 の多重 DO ループの一重化
do ij = 1, 10*20
  a(ij,1) = b(ij,1) + c(ij,1)
enddo
```

図 4.3-2 多重 DO ループの一重化の例

(2) コンパイラによる DO ループの一重化

多重 DO ループは，コンパイラによって自動的に一重化される場合がある．4.3.1.(1)であげた図 4.3-1 から図 4.3-2 への変形は，コンパイラが自動的に行うものである．コンパイラが変換可能なのは，多重ループが以下の条件を満たす場合である．

① 密な多重 DO ループ(注)である
② 指標変数を含む添字式が，その配列の次元の下限から上限までの値をとる
③ 各 DO ループの指標変数は，ループ中に現れる配列要素の添字中に左から連続して同じ形式かつ同じ順序で現れている
④ DO ループ中に現れる配列要素の各添字式は異なるループの指標変数を含まない
⑤ 一重化しても定義・引用の順序関係が正しく保たれる
⑥ ループ中に現れるすべての配列はポインタでも形状引継ぎ配列でもない

(注)密な多重 DO ループとは，直接の入れ子関係をなす DO ループのうち，外側ループの DO 文と内側ループの DO 文の間および内側ループの ENDDO 文と外側ループの ENDDO 文の間に，実行文が現れないもの(およびそれと等価な構造の多重ループ)．

(3) 自動的に一重化されない多重 DO ループのチューニング

DO ループ中にポインタや形状引継ぎ配列がある場合は上述⑥の条件を満たさないため，コンパイラによる自動的なループの一重化は行われない．このような場合，ループはコンパイル時にオプション「-pvctl collapse」(注 1)を指定することにより一重化することができる．

この他，自動的に一重化されない例として，図 4.3-3 のように DO ループの範囲の外に冗長な領域を持つ配列が含まれる場合をあげられる．ループの指標変数が配列 work の下限から上限までの値をとらず，上述②の条件を満たさないため，ループの一重化は行われない．図 4.3-4 のように配列 work の冗長な領域をなくすことで，コンパイラにより自動的に一重化されるようになる．なお，図 4.3-3 と図 4.3-4 は，コンパイラによる編集リスト(注 2)の形式で記載している．

```
!オリジナル
42:              subroutine sample_f(a,b,work,p)
43:                implicit none
44:                include 'parameter.h'
45:                real*8,intent(in)  :: a(imax,jmax,kmax,3)
46:                real*8,intent(in)  :: b(imax,jmax,kmax,3)
47:                real*8,intent(in)  :: work(imax+1,jmax+1,kmax+1,3)
48:                real*8,intent(out) :: p(imax,jmax,kmax)
49:                integer :: i,j,k
50:
51: +------>        do k=1,kmax
52: |+----->          do j=1,jmax
53: ||V---->            do i=1,imax
54: |||                  p(i,j,k)=work(i,j,k,1)*(a(i,j,k,1)+b(i,j,k,1)) &
55: |||                           +work(i,j,k,2)*(a(i,j,k,2)+b(i,j,k,2)) &
56: |||                           +work(i,j,k,3)*(a(i,j,k,3)+b(i,j,k,3))
57: ||V----            enddo
58: |+-----          enddo
59: +-------        enddo
```

図 4.3-3　多重 DO ループが一重化されないコードの例

```
!チューニング
42:              subroutine sample_f(a,b,work,p)
43:                implicit none
44:                include 'parameter.h'
45:                real*8,intent(in)  :: a(imax,jmax,kmax,3)
46:                real*8,intent(in)  :: b(imax,jmax,kmax,3)
47:                real*8,intent(in)  :: work(imax,jmax,kmax,3)
48:                real*8,intent(out) :: p(imax,jmax,kmax)
49:                integer :: i,j,k
50:
51: W------>        do k=1,kmax
52: |*----->          do j=1,jmax
53: ||*---->            do i=1,imax
54: |||                  p(i,j,k)=work(i,j,k,1)*(a(i,j,k,1)+b(i,j,k,1)) &
55: |||                           +work(i,j,k,2)*(a(i,j,k,2)+b(i,j,k,2)) &
56: |||                           +work(i,j,k,3)*(a(i,j,k,3)+b(i,j,k,3))
57: ||*----            enddo
58: |*-----          enddo
59: W------        enddo
```

図 4.3-4　多重 DO ループが一重化されないコードのチューニング例

(注 1)「-pvctl collapse」は，次元数が 2 以上の配列式，または多重 DO ループ中に現われる形状引き継ぎ配列，ポインタ配列の全要素がメモリ上の連続領域にあると仮定し，一重化を行うことを指示するコンパイルオプションである．

(注 2)編集リストとは，ソースコードと共に，ループと配列式のベクトル化情報，手続きのインライン展開情報，ループと配列式の並列化情報を出力する機能である．以下に編集リストに用いられる記号の意味を示す．

V：ベクトル化されたループ・配列式
W：一重化されてベクトル化されたループ・配列式
P：並列化されたループ
Y：並列化され，かつベクトル化されたループ
I：インライン展開された手続き
+：最適化されなかったループ

(4) 性能評価

(3)の図 4.3-3, 図 4.3-4 で示したチューニング前後のコードについて, 問題サイズを「imax= 50, jmax=200, kmax=1000」として, SX-9 1CPU を用いて性能評価を行う. SX-9 では, 性能情報として PROGINF(注 1)と FTRACE(注 2)がある. PROGINF からはプログラム全体の性能情報が取得でき, FTRACE からは手続(サブルーチンや関数)ごとの性能情報を取得することができる.

① PROGINF

図 4.3-5, 図 4.3-6 はチューニング前後の PROGINF である. 実効性能を比較するため, MFLOPS(1 秒間に実行された浮動小数点演算数を 100 万単位で示した値)の項目を見る. チューニング前は8,578MFLOPSであるのに対し, チューニング後には20,063MFLOPSとなっており, 実効性能が 2.33 倍向上していることが分かる. また, A.V.Length(平均ベクトル長(注 3))の項目を見ると, チューニング前は 49.9 であるのに対し, チューニング後は 252.3 となっている. チューニングにより多重 DO ループが一重化されることで, コード全体の平均ベクトル長が長くなっていることを確認できる.

```
    ******  Program Information  ******
Real Time (sec)                :          4.699699
User Time (sec)                :          4.672104
Sys  Time (sec)                :          0.023863
Vector Time (sec)              :          4.667600
Inst. Count                    :       3615133579.
V. Inst. Count                 :       1806406030.
V. Element Count               :      90320127249.
FLOP Count                     :      40080201114.
MOPS                           :      19718.922095
MFLOPS                         :       8578.619208
A. V. Length                   :         49.999904
V. Op. Ratio (%)               :         98.036742
Memory Size (MB)               :        960.031250
MIPS                           :        773.769929
I-Cache (sec)                  :          0.000384
O-Cache (sec)                  :          0.000478
Bank Conflict Time
  CPU Port Conf. (sec)         :          0.053537
  Memory Network Conf. (sec)   :          3.597606
```

図 4.3-5 チューニング前の PROGINF

```
    ******  Program Information  ******
Real Time (sec)                :          2.028015
User Time (sec)                :          1.997679
Sys  Time (sec)                :          0.025952
Vector Time (sec)              :          1.993263
Inst. Count                    :        695335689.
V. Inst. Count                 :        357973030.
V. Element Count               :      90320127249.
FLOP Count                     :      40080201166.
MOPS                           :      45381.410080
MFLOPS                         :      20063.384140
A. V. Length                   :        252.309866
V. Op. Ratio (%)               :         99.627871
Memory Size (MB)               :        960.031250
MIPS                           :        348.071782
I-Cache (sec)                  :          0.000325
O-Cache (sec)                  :          0.000416
Bank Conflict Time
  CPU Port Conf. (sec)         :          0.010343
  Memory Network Conf. (sec)   :          0.779060
```

図 4.3-6 チューニング後の PROGINF

② FTRACE

図 4.3-7, 図 4.3-8 はチューニング前後の FTRACE の情報である. ①PROGINF と同様に, 実効性能と平均ベクトル長を確認することができるが, FTRACE ではこれらの情報を手続(サブルーチンや関数)単位で取得することができる.

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
sample_f	500	4.638(99.5)	9.276	19794.6	8624.8	98.04	50.0	4.637	0.000	0.000	0.051	3.577
tp_sample	1	0.022(0.5)	22.060	14752.3	3635.5	98.33	50.0	0.022	0.000	0.000	0.000	0.015
total	501	4.660(100.0)	9.301	19770.7	8601.2	98.04	50.0	4.659	0.000	0.000	0.051	3.593

図 4.3-7 チューニング前の FTRACE 情報

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
sample_f	500	1.959(98.9)	3.918	46114.9	20420.2	99.63	256.0	1.959	0.000	0.000	0.011	0.715
tp_sample	1	0.022(1.1)	22.036	14749.9	3639.4	98.45	50.0	0.022	0.000	0.000	0.000	0.015
total	501	1.981(100.0)	3.954	45765.9	20233.5	99.63	252.3	1.980	0.000	0.000	0.011	0.730

図 4.3-8 チューニング後の FTRACE 情報

(注 1)プログラム実行解析情報:プログラム全体の性能情報が標準エラー出力ファイルに出力される. 以下に PROGINF 機能で取得可能な情報を示す.

Real Time	:経過時間(プログラム実行に要した時間)
User Time	:ユーザ時間(プログラム実行に要した CPU 時間の内, ユーザルーチンが実行に要した時間)
Sys Time	:システム時間(プログラム実行に要した CPU 時間の内, システムルーチンが実行に要した時間)
Vector Time	:ベクトル命令実行時間
Inst. Count	:全命令実行数
V. Inst. Count	:ベクトル命令実行数
V. Element Count	:ベクトル命令で実行された演算要素の個数
FLOP Count	:実行された浮動小数点演算の個数
MOPS	:1 秒間に実行された演算数を 100 万単位で示した値
MFLOPS	:1 秒間に実行された浮動小数点演算数を 100 万単位で示した値
A. V. Length	:平均ベクトル長(注 3 参照)
V. Op. Ratio	:全実行命令の個数の内, ベクトル命令の占める割合
Memory Size	:メモリ使用量
MIPS	:1 秒間に実行された命令数を 100 万単位で示した値
I-Cache	:命令キャッシュミスにより発生した合計時間
O-Cache	:オペランドキャッシュミスにより発生した合計時間
Bank Conflict Time	:バンクコンフリクト時間
CPU Port Conf.	:CPU ポート競合時間
Memory Network Conf.	:メモリネットワーク競合時間

(注 2)性能解析情報:コンパイル時にオプション「-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．以下にFTRACE機能で取得可能な情報を示す．

PROC.NAME	:手続名
FREQUENCY	:手続の呼び出し回数
EXCLUSIVE TIME	:手続の実行に要した占有のCPU時間と，手続全体の実行に要したCPU時間に対する比率
AVER.TIME	:手続の1回の実行に要した平均CPU時間
MOPS	:1秒間に実行された演算数を100万単位で示した値
MFLOPS	:1秒間に実行された浮動小数点演算数を100万単位で示した値
V.OP RATIO	:ベクトル演算率
AVER V.LEN	:平均ベクトル長(注3 参照)
VECTOR TIME	:ベクトル命令実行時間
I-CACHE MISS	:命令キャッシュミスにより発生した合計時間
O-CACHE MISS	:オペランドキャッシュミスにより発生した合計時間
BANK CONFLICT	:バンクコンフリクト時間
CPU PORT	: CPUポート競合時間
NETWORK	: メモリネットワーク競合時間

(注3)SX-9の性能情報の一つに，平均ベクトル長がある．SX-9は一つのベクトル命令で最大256組の演算要素を処理することができるので，例えばDOループの長さが1000の場合，256, 256, 256, 232という4つの処理単位にDOループを分割して処理する．この場合，平均ベクトル長は250になる．

4.3.2 多重DOループのアンローリングによる高速化

(1) 多重DOループのアンローリングの概要

DOループのアンローリングとは，ループ本体の演算をn倍にし，繰り返し数を1/nにする処理である．このとき nをアンローリングの段数という．

図4.3-9の外側のjのループについて，2段のアンローリングをしたのが図4.3-10である．外側のjのループは増分2とし，内側のiのループ内に j+1 の演算を追加して，内側ループの演算を2倍にしている．これにより，図 4.3-10 の①の x(i)のストアと②の x(i)のロードが不要になる．つまり，図 4.3-10は図 4.3-11 のように変形できる．よって，多重 DO ループ全体で考えると，x(i)のロード回数，ストア回数がアンローリング前の半分になる．

多重 DO ループのアンローリングでは，一般に，ベクトル化されない外側ループについてアンローリングを行うことで性能向上が期待できる．特にロード・ストア回数が減少するような場合には，大きな効果が得られることが多い．

```
!多重 DO ループ
do  j=1,100
    do  i=1,n
        x(i)=x(i)+a(i,j)*b(j)
    enddo
enddo
```

図 4.3-9　多重 DO ループの例

```
!図 4.3-9 の多重 DO ループを，外側ループについて 2 段でアンローリング
do  j=1,99,2
    do  i=1,n
        x(i)=x(i)+a(i,j)*b(j)          ・・・①
        x(i)=x(i)+a(i,j+1)*b(j+1)      ・・・②
  enddo
enddo
```

図 4.3-10　外側 DO ループのアンローリング後

```
!図 4.3-10 の x(i)のロード・ストアを削減した結果
do  j=1,99,2
    do  i=1,n
        x(i)=x(i)+a(i,j)*b(j)+a(i,j+1)*b(j+1)
  enddo
enddo
```

図 4.3-11　x(i)のロード・ストア削減の結果

(2)　差分コードのアンローリング

図 4.3-12 は，差分コードのチューニング例である．多重 DO ループにおいてコンパイラが外側ループのアンローリングを行うように，コンパイラ指示行 OUTERUNROLL(注 1)を挿入している．

```
!チューニング
 :
47: +------>      do k=2,kmax-1
48: |             !CDIR OUTERUNROLL=4
49: |+----->        do j=2,jmax-1
50: ||V---->           do i=2,imax-1
51: |||                s0=c(i,j,k,1)*(a(i+1,j+1,k)-a(i+1,j-1,k) &
52: |||                              -a(i-1,j+1,k)+a(i-1,j-1,k)) &
53: |||                  +c(i,j,k,2)*(a(i,j+1,k+1)-a(i,j-1,k+1) &
54: |||                              -a(i,j+1,k-1)+a(i,j-1,k-1)) &
55: |||                  +c(i,j,k,3)*(a(i+1,j,k+1)-a(i-1,j,k+1) &
56: |||                              -a(i+1,j,k-1)+a(i-1,j,k-1))
57: |||                ss=s0*b(i,j,k)
58: |||                p(i,j,k)=a(i,j,k)+omega*ss
59: ||V----           enddo
60: |+-----         enddo
61: +------       enddo
 :
```

図 4.3-12　差分コードのチューニング例

図 4.3-13 は図 4.3-12 のチューニング例について，アンローリング部分の変形結果を変形リスト(注 2)より抜粋したものである．

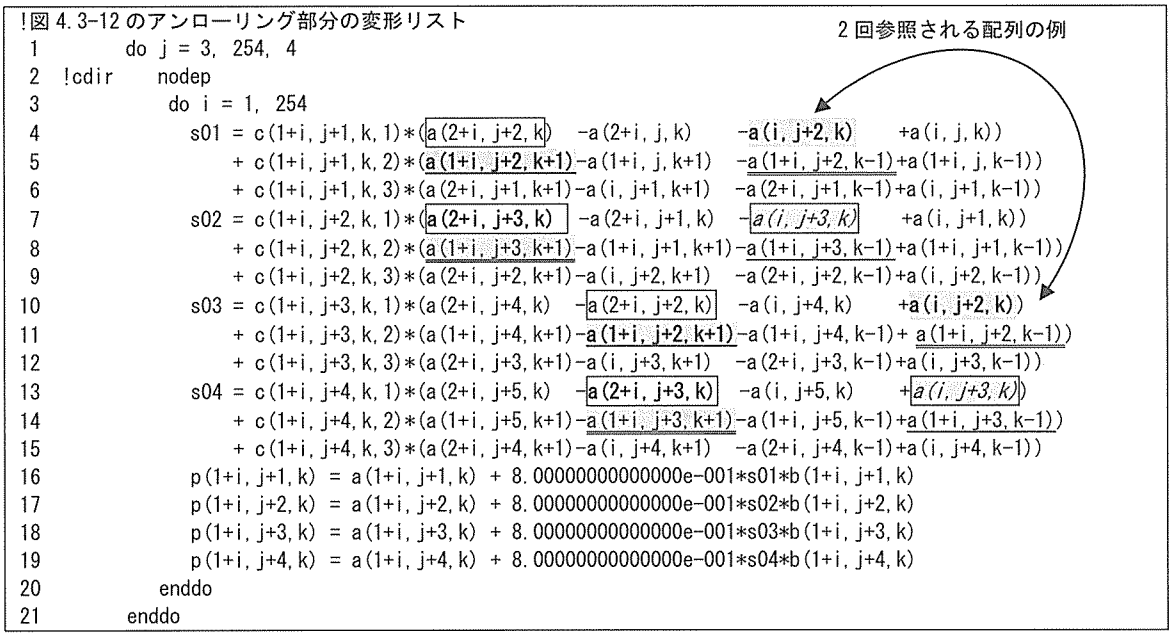

図 4.3-13 アンローリング後の変形リスト

この 4 段のアンローリングの例では，配列 a，b，c は全部で 68 個の要素がロードされるが，そのうち配列 a は 8 個の要素が 2 回参照される(注 3)．重複する部分は再度ロードされることが無いため，DO ループ内のロード回数が 68 回から 60 回に減る．同様に，8 段でアンローリングした場合は，配列 a, b, c は全部で 136 個の要素がロードされるが，そのうち配列 a は 24 個の要素が 2 回参照されるため，DO ループ内のロード回数が 136 回から 112 回に減る．

(注 1) 直後の外側 DO ループに対し，強制的に n 段のアンローリングを行うことを指定する指示行である．これを指定した場合，コンパイラはループ内のデータの依存関係をチェックせずにアンローリングを行う．なお OUTERUNROLL の段数は，指定した値以下の 2 のべき乗の値となる．

(注 2) 変形リスト:コンパイラが行った最適化の結果，実際に生成されたオブジェクトコードの構造をソースコードのイメージで表示したリストである．

(注 3) 図 4.3-13 では，変形リストを抜き出して行番号を追記し，2 回参照される配列がわかりやすいよう，配列ごとに形式を変えて強調表記している．例えば，行番号 4 と 10 の矢印で示した a(i,j+2,k)の配列は a(i,j+2,k)と表記している．

(3) 性能評価

(2)の図 4.3-12，図 4.3-13 で示しているチューニング前後のコードについて，問題サイズを「imax=256, jmax=256, kmax=256」とし，SX-9 1CPU を用いて性能評価を行う．表 4.3-1 が性能測定結果である．4 段でアンローリングする場合には，チューニング前に比べて 1.16 倍，8 段でアンローリングする場合には，1.30 倍性能向上していることが分かる．

表 4.3-1 アンローリングの性能比較

	性能
チューニング前	16.29GFLOPS
アンローリング後(4 段)	18.91GFLOPS
アンローリング後(8 段)	20.21GFLOPS

(4) アンローリング段数による性能の違い

(3)での評価結果からもわかるとおり，アンローリング段数によって性能に差が生じている．図 4.3-14 は，(2)の差分コードについて，段数を 2, 4, 8, 16, 32 と変えて性能測定している結果である．

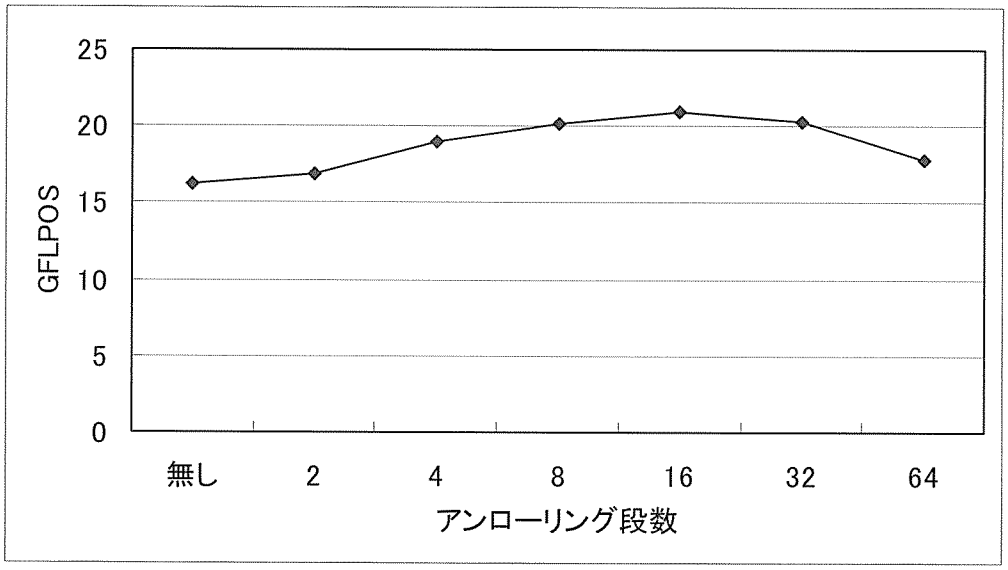

図 4.3-14 アンローリング段数による性能差

一般に，アンローリング段数を増やすと性能が向上する．しかし，段数が増えるに伴いループ中の中間変数が増加する．そのため段数を増やしすぎると途中で中間変数を保存するレジスタが足りなくなり，性能が劣化する．図 4.3-14 のとおり，(2)の差分コードの例では，16 段でアンローリングすると最も性能が高くなる．最適なアンローリング段数はコードにより異なる．

(5) アンローリングの自動/手動に関する補足

外側の DO ループをアンローリングすることで，内側の DO ループのベクトル演算パイプラインを有効に利用できる場合や，ロード・ストア回数を減らす効果がある場合，コンパイラは自動的にアンローリングを行う．コンパイラによって自動的にアンローリングされる場合でも，デフォルトの段数（通常 4 段）以外で実行すると更に性能向上が得られる場合がある．多重ループでは，ロード・ストア回数が最も削減される次元でアンロールするのが有効である．

4.3.3 IF 文を含む DO ループのチューニング

(1) IF 文を含む DO ループのベクトル処理の概要

図 4.3-15 のように IF 文を含む DO ループは，つぎのように処理される．

① 条件式「flag(n).eq.1」の真偽に応じて，マスクベクトルを作成

② 条件式の真偽にかかわらず，b(n)+c(n) を演算

③ マスクベクトルが真である要素のみ，b(n)+c(n)の演算結果を a(n)に代入

このように，IF 文を含まない DO ループに比べ，条件分岐のための処理が加わるので，分岐の数が多いと性能劣化の要因となる．

```
real,dimension(nsp)::a,b,c
integer::flag(nsp)

do n=1,nsp
  if(flag(n).eq.1)then
    a(n)=b(n)+c(n)
  endif
enddo
```

図 4.3-15 IF 文を含んだ DO ループの例

なお，IF 文を含んでいても，図 4.3-16 のようにループの繰り返しの間，条件式の結果が不変の場合，コンパイラは IF 文を DO ループの外に出してベクトル化するので，IF 文のオーバヘッドは発生しない．

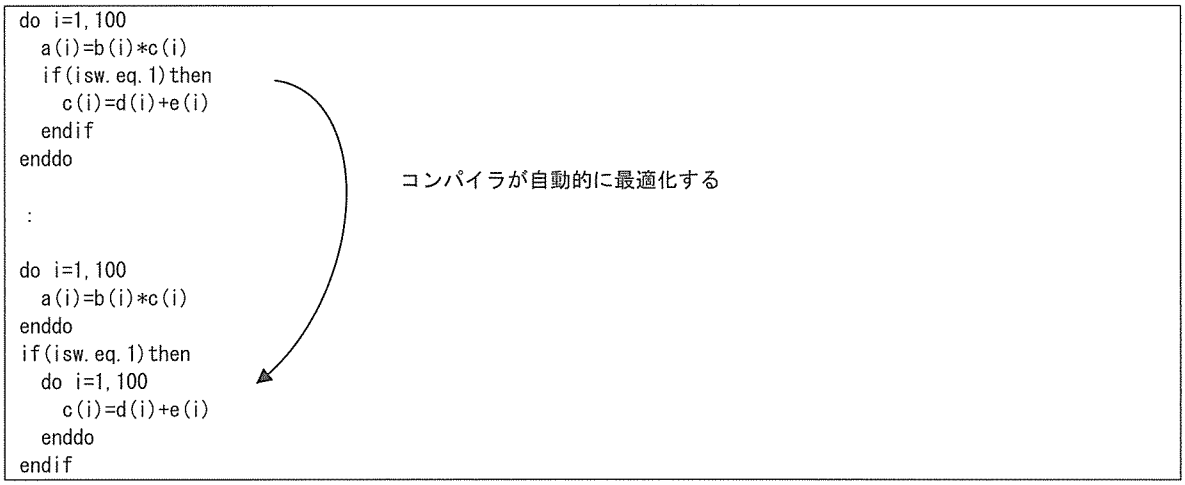

図 4.3-16 IF 文が DO ループの外に展開される変形イメージ

(2) マスクベクトルを用いてベクトル化しているループの性能

図 4.3-17 の DO ループは，マスクベクトルを用いてベクトル化するので TRUE，FALSE のどちらの場合も演算が実行される．演算結果は，マスクベクトルを用いて TRUE に対応する要素のみ代入される．このため，flag_v の真の割合(以下，真率)に関係なく，DO ループの処理時間はほぼ一定になる．

```
do n=1,nsp
  if(flag_v(n) .eq. 1) then
    tmpx3(n)=(tmpx1(n)-tmpx2(n))*2.0d0
    tmpy3(n)=(tmpy1(n)-tmpy2(n))*2.0d0
    tmpz3(n)=(tmpz1(n)-tmpz2(n))*2.0d0
  endif
enddo
```

図 4.3-17 一般的な IF 文を含む DO ループ

図 4.3-18 は，以下の 3 パターンで図 4.3-17 の DO ループを SX-9 1CPU で実行するときの性能値である．それぞれの実行時間がほとんど変わらないことがわかる．

- Rfv_vt　　:flag_v の値が全て TRUE の場合
- Rfv_vf　　:flag_v の値が全て FALSE の場合
- Rfv_50　　:flag_v の値がランダムに TRUE/FALSE となる場合

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
Rfv_vt	30	0.757(0.3)	25.237	48061.8	15954.8	99.59	256.0	0.757	0.000	0.000	0.006	0.327
Rfv_vf	30	0.755(0.3)	25.174	48182.2	15994.8	99.59	256.0	0.755	0.000	0.000	0.000	0.327
Rfv_50	30	0.758(0.3)	25.262	48015.4	15939.4	99.59	256.0	0.758	0.000	0.000	0.014	0.329

図 4.3-18 FTRACE 情報の一覧

(3) 事例1:IF 文をループの外に移動するチューニング

図 4.3-19 の多重ループは, TRUE 側の演算量が FALSE 側の演算量よりも大きい IF 文の例である. マスクベクトルを用いるベクトル化は, 条件式の結果に関わらず, どちらの演算も実行しているため, 本来必要な演算数より多くなる. 以下に, ループの入れ替えによって IF 文をループの外に移動するチューニングの例を示す.

```
!オリジナル
153: +------>     do k=1,nz
154: |+----->       do j=1,ny
155: ||V---->         do i=1,nx
156: |||               if(lc(i,k).eq.0) then
157: |||                 p(1,i,j,k) = ff*va(1,i,j,k)*(f1*vb(i,j,k)+gv10) + d1*c(i,j,k)
158: |||                 p(2,i,j,k) = ff*va(2,i,j,k)*(f2*vb(i,j,k)+gv10) + d2*c(i,j,k)
159: |||                 p(3,i,j,k) = ff*va(3,i,j,k)*(f3*vb(i,j,k)+gv10) + d3*c(i,j,k)
160: |||               else
161: |||                 p(1,i,j,k) = d1*c(i,j,k)
162: |||                 p(2,i,j,k) = d2*c(i,j,k)
163: |||                 p(3,i,j,k) = d3*c(i,j,k)
164: |||               endif
165: ||V----           enddo
166: |+-----         enddo
167: +------        enddo
```

図 4.3-19 TRUE 側の演算量が FALSE 側の演算量よりも大きい IF 文のコード

この IF 文の条件式「lc(i,k).eq.0」はループ変数 j に依存しておらず, ループ内の配列に依存関係がない. このため DO ループ j を IF 文の内側に移動できる.

チューニング後のコードを図 4.3-20 に示す. IF 文は DO ループ j よりも外側にあり, マスクベクトル処理とならない. このチューニングで IF 文の条件式により内側の DO ループが選択され, 真の時だけ実行される.

```
!チューニング
153: +------>     do k=1,nz
154: |+----->         do i=1,nx
155: ||                 if(lc(i,k).eq.0) then
156: ||V---->         do j=1,ny
157: |||                   p(1,i,j,k) = ff*va(1,i,j,k)*(f1*vb(i,j,k)+gv10) + d1*c(i,j,k)
158: |||                   p(2,i,j,k) = ff*va(2,i,j,k)*(f2*vb(i,j,k)+gv10) + d2*c(i,j,k)
159: |||                   p(3,i,j,k) = ff*va(3,i,j,k)*(f3*vb(i,j,k)+gv10) + d3*c(i,j,k)
160: ||V----          enddo
161: ||                 else
162: ||V---->         do j=1,ny
163: |||                   p(1,i,j,k) = d1*c(i,j,k)
164: |||                   p(2,i,j,k) = d2*c(i,j,k)
165: |||                   p(3,i,j,k) = d3*c(i,j,k)
166: ||V----          enddo
167: ||                 endif
168: |+-----           enddo
169: +------        enddo
```

図 4.3-20 IF 文をループの外に移動したコード

チューニング前後の各ケースについてSX-9 1CPUを用いて性能評価を行う. 問題サイズは「nx=256, ny=256, nz=128」としている. IF 文の条件式の真率による評価を行うため, チューニング前後で真率をそれぞれ以下に設定する.

- Res_vt : IF 文の条件式が全て TRUE の場合
- Res_vf : IF 文の条件式が全て FALSE の場合
- Res_25 : IF 文の条件式が真率 25%となる場合
- Res_50 : IF 文の条件式が真率 50%となる場合
- Res_75 : IF 文の条件式が真率 75%となる場合

!オリジナル

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
Res_vt	30	0.145(11.8)	4.836	67836.6	36428.7	99.74	256.0	0.145	0.000	0.000	0.053	0.035
Res_vf	30	0.145(11.8)	4.827	67955.3	36492.4	99.74	256.0	0.145	0.000	0.000	0.052	0.037
Res_25	30	0.163(13.2)	5.421	60511.5	32495.0	99.74	256.0	0.163	0.000	0.000	0.067	0.054
Res_50	30	0.172(13.9)	5.719	57358.6	30801.9	99.74	256.0	0.172	0.000	0.000	0.072	0.062
Res_75	30	0.163(13.2)	5.427	60447.5	32460.7	99.74	256.0	0.163	0.000	0.000	0.068	0.049

図 4.3-21 チューニング前の FTRACE 情報

!チューニング

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
Res_vt	30	0.123(13.9)	4.116	61321.9	36687.8	99.71	256.0	0.123	0.000	0.000	0.064	0.044
Res_vf	30	0.095(10.6)	3.151	18804.9	7987.7	99.11	256.0	0.094	0.000	0.000	0.045	0.060
Res_25	30	0.105(11.8)	3.514	30567.0	16091.8	99.46	256.0	0.105	0.000	0.000	0.050	0.057
Res_50	30	0.115(13.0)	3.844	40563.3	22932.7	99.60	256.0	0.115	0.000	0.000	0.054	0.055
Res_75	30	0.120(13.4)	3.983	51256.7	30021.8	99.67	256.0	0.119	0.000	0.000	0.059	0.049

図 4.3-22 チューニング後の FTRACE 情報

チューニング前は, 実行時間にわずかな差が出ているが, 全ての場合で全浮動小数点演算数(MFLOPS×時間)の変化がない. このことから, IF 文の条件式の真率に関係なく TRUE/FALSE 両方の演算がなされていることがわかる.

チューニング後は, 全浮動小数点演算数は IF 文の条件式の真率よって増減している. 演算密度の違いで MFLOPS 値が低下している場合もあるが, どの場合でも処理時間の短縮が確認できる.

(4) 事例2:IF 文を他の処理に置き換えるチューニング

図 4.3-23 の例は, 粒子コードに見られる周期的境界条件で位置情報の補正を行うコードである. DO ループ中に多数の IF 文があるため, 多くのマスク付きベクトル演算を行い, 演算時間が長くなっている. この根本的な解決として IF 文に代わる処理への置き換えを実施する.

```
!チューニング前
54: V------>      do n=1,nsp
55: |               if(pa(n,1) < 0.0d0) pa(n,1)=pa(n,1)+xdim
56: |               if(pa(n,1) >= xdim) pa(n,1)=pa(n,1)-xdim
57: |               if(pa(n,2) < 0.0d0) pa(n,2)=pa(n,2)+ydim
58: |               if(pa(n,2) >= ydim) pa(n,2)=pa(n,2)-ydim
59: |               if(pa(n,3) < 0.0d0) pa(n,3)=pa(n,3)+zdim
60: |               if(pa(n,3) >= zdim) pa(n,3)=pa(n,3)-zdim
61: V------       enddo
```

図 4.3-23 IF 文による周期的境界条件のコード

図 4.3-23 のコードは, 位置情報が各方向の最小値から最大値の範囲に収まっているかを, IF 文の

条件式で判定している. 収まっていない場合は, 位置情報に各辺の長さを加減して周期的境界条件を満たす補正がなされる.

図 4.3-24 にチューニング後のコードを示す. ここでは, mod 関数を使用して周期的境界条件を満たす同等な補正を行っている. 全要素一律に各辺の長さを加算し, その長さで割った余りを求めることで周期的境界条件が満たされ, IF 文による場合分けが不要となる. ただし, 周期内の条件にある位置は, mod 関数による補正計算が行われるため, チューニング前のコードと比べて若干の誤差が生じる. どちらのコードもループ長は同じである.

```
!チューニング後
54: V-------->      do n=1,nsp
55: |                 pa(n,1)=dmod(pa(n,1)+xdim,xdim)
56: |                 pa(n,2)=dmod(pa(n,2)+ydim,ydim)
57: |                 pa(n,3)=dmod(pa(n,3)+zdim,zdim)
58: V-------        enddo
```

図 4.3-24 mod 関数による周期的境界条件のコード

図 4.3-23 と図 4.3-24 の各場合について SX-9 1CPU を用いて性能評価を行う. 問題サイズは「nx=256, ny=128, nz=64, ns=32」としている. 要素数 nsp は nx, ny, nz, ns の積となり, 各方向の最大値を実数型変数 xdim, ydim, zdim で与えている.

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
sample_f	30	9.557(85.8)	318.582	6327.7	1895.8	99.87	256.0	9.557	0.000	0.000	0.021	8.658

図 4.3-25 チューニング前の FTRACE 情報

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
sample_f	30	1.720(51.5)	57.345	52707.6	17554.0	99.91	256.0	1.720	0.000	0.000	0.004	0.029

図 4.3-26 チューニング後の FTRACE 情報

図 4.3-25 と図 4.3-26 がそれぞれチューニング前後の性能値で, ベクトル演算率やベクトル長に大きな差異は見られない. チューニング前は配列 pa のロード・ストア回数が多く, 条件分岐のオーバヘッドで待ち時間も発生し, 「BANK CONFLICT/NETWORK」が大きくなっている. チューニング後は mod 関数を用い IF 文の条件分岐が取り除かれたことで, ベクトル演算の密度が上がり MOPS および MFLOPS の値が向上している.

4.3.4 DO ループ内の作業配列の削除による高速化

(1) 作業配列の削除の概要

一般にプログラムは入力データと出力結果の他にも, 途中経過を作業配列に保存する場合がある. しかし, 不必要な作業配列の利用はメモリ負荷を増加させ, 演算性能を低下させることになる.

図 4.3-27 の例は, ループ内でのみ使用する途中の演算結果 a(n)+b(n)を作業配列 work(n)に保存している. この途中の演算結果をその後の処理で利用することがなければ, 作業配列 work(n)を使わずスカラ変数に書き換え, 作業配列への保存を回避できる. このようにすることで, 途中の演算結果は作業レジスタに保存され, メモリ負荷が下がり演算性能が向上する.

```
subroutine sample_f(a,b,p,nsize)
  real*8 :: work(nsize)
    :
  do n=1,nsize
    work(n)=a(n)+b(n)
    p(n)=work(n)*0.5
  enddo
    :
end subroutine sample_f
```

ループ内でのみ使用する値は，
配列に保存せずスカラ変数を使う．

図 4.3-27 作業配列からスカラ変数に書き換えることができる例

(2) 作業配列を削減するチューニングの事例

図 4.3-28 のサブルーチン sample_f1，sample_f2 は，プログラム中の複数の箇所で同じ演算となる部分を括りだし，共通部品としてサブルーチンにしたものである．このプログラムは，3 次元配列のデータを 1 次元作業配列にコピーしサブルーチン sample_f1 へ渡し，サブルーチン sample_f1，sample_f2 で計算し，1 次元作業配列の結果を 3 次元配列に書き戻している．

```
!オリジナル
 34: |+------>       do k=1,kmax
 35: ||+----->       do j=1,jmax
 36: |||
 37: |||V--->          do i=1,imax
 38: ||||               wi(1,i)=a(1,i,j,k)
 39: ||||               wi(2,i)=a(2,i,j,k)
 40: ||||               wi(3,i)=a(3,i,j,k)
 41: ||||               wv(i)  =b(i,j,k)
 42: |||V---            enddo
 43: |||
 44: |||               call sample_f1(imax,wi,c ,wt)
 45: |||               call sample_f2(imax,wv,wt,wo)
 46: |||
 47: |||V--->          do i=1,imax
 48: ||||               p(i,j,k)=wo(i)
 49: |||V---            enddo
 50: |||
 51: ||+----          enddo
 52: |+------          enddo

 85:             subroutine sample_f1(lmax,wi,wc,wt)
  :
 97: V------>       do l=2,lmax-1
 98: |                wt(l)=wc(1)*(wi(1,l+1)-wi(1,l-1)) &
 99: |                     +wc(2)*(wi(2,l+1)-wi(2,l-1)) &
100: |                     +wc(3)*(wi(3,l+1)-wi(3,l-1))
101: V------        enddo

106:            subroutine sample_f2(lmax,wv,wt,wo)
  :
118: V------>      do l=2,lmax-1
119: |              wo(l)=wt(l)*wv(l)
120: V------       enddo
```

図 4.3-28 作業配列を削減前のコード

図 4.3-29 のチューニングは，作業配列 wi,wv と元の入力配列 a,b の 1 列が同じ値を参照していることに着目する．この場合，作業配列 wi,wv を使わないで配列 a,b の 1 列を直接引き渡すことができる．同様に引数の作業配列 wo と出力配列 p の 1 列が同じ値を更新しているので，出力データも作業配列 wo を使わないで配列 p へ直接出力できる．このようにサブルーチン呼び出し前後に作業配列 wi，wv，wo を使用しないでデータを引き渡すことができるため，作業配列へのメモリアクセス回数が削減でき

る.

```
!チューニング①
 58: |+-----> do k=1,kmax
 59: ||+----> do j=1,jmax
 60: |||
 61: |||        call sample_f1(imax,a(1,1,j,k),c ,wt      )
 62: |||        call sample_f2(imax,b(1,j,k)  ,wt,p(1,j,k))
 63: |||
 64: ||+-----  enddo
 65: |+------  enddo

 85:          subroutine sample_f1(lmax,wi,wc,wt)
 :
 97: V------>   do l=2,lmax-1
 98: |            wt(l)=wc(1)*(wi(1,l+1)-wi(1,l-1)) &
 99: |                 +wc(2)*(wi(2,l+1)-wi(2,l-1)) &
100: |                 +wc(3)*(wi(3,l+1)-wi(3,l-1))
101: V------    enddo

106:         subroutine sample_f2(lmax,wv,wt,wo)
 :
118: V------>   do l=2,lmax-1
119: |            wo(l)=wt(l)*wv(l)
120: V------    enddo
```

図 4.3-29 [第 1 段階]入力データと出力データを引数にする修正

図 4.3-30 の更なるチューニングは, サブルーチン sample_f1, sample_f2 を 1 つのサブルーチン sample_f にまとめ, これらのサブルーチンそれぞれにあった DO ループを融合している. このループ融合によって, sample_f1 から sample_f2 に受け渡している途中の結果を, 作業配列 wt を使わずに演算処理できる.

```
!チューニング②
 71: |+-----> do k=1,kmax
 72: ||+----> do j=1,jmax
 73: |||
 74: |||        call sample_f(imax,a(1,1,j,k),b(1,j,k),c,p(1,j,k))
 75: |||
 76: ||+-----  enddo
 77: |+------  enddo

127:         subroutine sample_f(lmax,wi,wv,wc,wo)
 :
141: V------>   do l=2,lmax-1
142: |            s=wc(1)*(wi(1,l+1)-wi(1,l-1)) &
143: |             +wc(2)*(wi(2,l+1)-wi(2,l-1)) &
144: |             +wc(3)*(wi(3,l+1)-wi(3,l-1))
145: |            wo(l)=s*wv(l)
146: V------    enddo
```

図 4.3-30 [第 2 段階]サブルーチンにまたがる 2 つの DO ループを融合する修正

(3) 性能評価

チューニング前後の各場合について SX-9 1CPU を用いて性能評価を行う. (2)で示したコードを用い, 問題サイズは「imax=1000, jmax=50, kmax=50」としている. 図 4.3-31 は, チューニング前後の性能値である.

- case_000:図 4.3-28 チューニング前の結果
- case_001:図 4.3-29 チューニング[第 1 段階]での結果

- case_002:図 4.3-30 チューニング[第 2 段階]での結果

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
case_000	1000	5.841(100.0)	5.841	12542.9	3844.5	98.83	249.7	5.840	0.000	0.000	1.862	4.740
case_001	1000	1.821(100.0)	1.821	26312.6	12331.1	98.93	249.5	1.820	0.000	0.000	0.520	1.145
case_002	1000	1.064(100.0)	1.064	40150.0	21097.7	99.26	249.5	1.064	0.000	0.000	0.516	0.423

図 4.3-31 FTRACE 情報一覧

case_000→case_001→case_002 とチューニングの段階を経て,「BANK CONFLICT/ NETWORK」の時間が短縮されている. これは作業配列へのデータ転送が減っているためである. データ転送が削減されメモリ負荷が下がったことで, 結果的に演算器の待ち時間が減少し, MFLOPS 値が向上している.

4.3.5 組み込み関数 MAX/MIN の利用による高速化

(1) 組み込み関数 MAX/MIN の利用による高速化の概要

最大値・最小値を求める計算はマクロ演算を利用してベクトル化され, 図 4.3-32 に示すようにコンパイル診断メッセージに「Macro operation Max/Min」と表示される. また, 4.1 節で述べたとおり, SX-9 から MAX/MIN 関数がハードウェア命令化され, 最大値・最小値を求める計算時間の短縮が行われる. 本節では, 最大値を求めるコードでベクトル化される場合とベクトル化されない場合を示し, さらにハードウェア命令が使われた場合と, 使われない場合の性能の違いを示す.

```
    61  vec  (  26): Macro operation Max/Min.

 :(※中略)

57: W------>      do k=1,kmax
58: |*------>       do j=1,jmax
59: ||*----->         do i=1,imax
60: |||                 tmp=r1(i,j,k)*r2(i,j,k)
61: |||                 s=max(s,tmp)
62: ||*----           enddo
63: |*-----         enddo
64: W------       enddo
```

行番号 61 番で最大値・最小値のマクロ演算を適用するメッセージ

図 4.3-32 MAX/MIN 関数利用の例とコンパイル診断メッセージ

(2) 最大値を求めるコードの事例

次に示す図は, 3 種類の最大値を求めるコードで, 以下の特徴がある.

- コード①(図 4.3-33)は, 組み込み関数 MAX を利用して最大値を求める
- コード②(図 4.3-34)は, 大小関係を IF 文で判断し, 大きい値で変数 s を更新し最大値を求める
- コード③(図 4.3-35)は, ループの 1 回目で初期値を設定し, 2 回目以降のループで最大値を求める

コード①は, 組み込み関数 MAX を利用しているので SX-9 のハードウェア命令が使われベクトル化される. コード②はコンパイラが最大値を求める処理と認識できるのでベクトル化される. コード③は条件分岐が複雑であり, コンパイラが最大値を求める処理と認識できないのでベクトル化されない.

```
!コード①
131:              subroutine sample_b1(r1,r2,s)
:
143:                s=-999d0
144: V------->      do i=1,imax*jmax*kmax
145: |                tmp=r1(i)*r2(i)
146: |                s=max(s,tmp)
147: V-------       enddo
```

図 4.3-33 最大値を求めるコード①

```
!コード②
154:              subroutine sample_b2(r1,r2,s)
:
166:                s=-999d0
167: V------->      do i=1,imax*jmax*kmax
168: |                tmp=r1(i)*r2(i)
169: |                if(tmp .gt. s) s=tmp
170: V-------       enddo
```

図 4.3-34 最大値を求めるコード②

```
!コード③
177:              subroutine sample_b3(r1,r2,s)
:
190:                lflag=.true.
191: +------->      do i=1,imax*jmax*kmax
192: |                tmp=r1(i)*r2(i)
193: |                if(lflag) then
194: |                  s=tmp
195: |                  lflag=.false.
196: |                else
197: |                  s=max(s,tmp)
198: |                endif
199: +-------       enddo
```

図 4.3-35 最大値を求めるコード③

(3) 性能評価

各コードを SX-9 1CPU を用いて性能評価を行う. 問題サイズは「imax*jmax*kmax=256^3」で十分に長いループ長としている.

表 4.3-2 3 種類コードの性能

	性能値
コード①	20,999MFLOPS
コード②	11,373MFLOPS
コード③	93MFLOPS

コード①は MAX/MIN 関数でベクトル化され SX-9 で強化されたハードウェア命令が使われるため, 高い性能を示している. コード②はベクトル化されるが SX-9 のハードウェア命令が使われてないため, コード①と比較して半分程度の性能である. コード③はベクトル化されていないので性能が出ていない. SX-9 で最大値・最小値を求める場合, 高い性能を得るには MAX/MIN 関数を使用してベクトル化することが必要である.

4.3.6 科学技術計算ライブラリASLの利用による高速化

(1) 科学技術計算ライブラリ利用の概要

数値計算などで利用している連立 1 次方程式, 固有値・固有ベクトル, 高速フーリエ変換(FFT), 特殊関数, 乱数などの機能は計算ライブラリとして準備されている. このような計算ライブラリをユーザが作成することはできるが, ハードウェアの性能を引き出すために複雑な最適化が必要であり, 高い性能を得るのは容易ではない. また一般の計算ライブラリは, 特定のハードウェアに向けての最適化は通常行われておらず, SX-9 の最大性能を得ることは難しい. SX-9 には科学技術計算ライブラリ ASL があり, これは広範な分野の数値シミュレーションプログラムの作成を強力に支援する計算ライブラリである. ASL はハードウェアの性能を最大限に引き出す最適化が行われており, ユーザはこれらのライブラリを使うことにより, SX-9 の性能を引き出すことが容易に行える.

これから科学技術計算ライブラリ ASL の FFT とフリーソフトウェアの FFT ライブラリとの性能を比較する. 比較対象のフリーソフトウェアは, NETLIB の FFTPACK, および筑波大学高橋大介氏作成の FFTE 4.1 としている. それぞれの FFT ライブラリは, 以下の URL からダウンロードできる.

FFTPACK	http://www.netlib.org/fftpack/
FFTE	http://www.ffte.jp/

(2) ASL と FFTPACK の性能評価

ここでは1次元複素数フーリエ変換を ASL のサブルーチン JFCOSI/JFCOSF と FFTPACK のサブルーチン ZFFTI/ZFFTF/ZFFTB を用いて比較する. それぞれの評価コードを図 4.3-36 と図 4.3-37 に示す. これらサブルーチンは周期2πの1周期を n 等分した n 個の複素数データが与えられたとき, n 個のデータに対する離散フーリエ変換(順変換と逆変換)を行う.

```
! 1次元複素数 FFT (ASL) の評価コード
    integer, parameter :: ld=2**26
    complex(kind=8) :: c(ld),trigs((3*ld)/2),wc(ld)
    integer :: ifax(128)
    real(kind=8) :: t1,t2,tm,perf(-1:1)
    do ii=11,26
     n=2**ii
     do isw=1,-1,-2
       do i=1,n
        c(i) = dcmplx(cos(sqrt(2.d0)*i),sin(sqrt(2.d0)*i))
       enddo

       call JFCOSI(n,ifax,trigs,ierr)
       call clock(t1)
       call JFCOSF(n,c,ld,isw,ifax,trigs,wc,ierr)
       call clock(t2)
       tm=(t2-t1)
       perf(isw)=5.d0*n*log(dble(n))/log(2.d0)/tm*1.d-9

     enddo
     write(6,*) n,perf(1),perf(-1)
    enddo
    end
```

ASL の FFT サブルーチン

図 4.3-36 1 次元複素数 FFT(ASL)の評価コード

```
! 1次元複素数FFT (FFTPACK) の評価コード
    integer, parameter :: ld=2**26
    complex(kind=8) :: w(ld*2+15),cx(ld)
    real(kind=8) :: t1,t2,tm,perf(-1:1)
    do ii=11,26
     n=2**ii
     do isw=1,-1,-2
       do i=1,n
        cx(i) = dcmplx(cos(sqrt(2.d0)*i),sin(sqrt(2.d0)*i))
       enddo

       call ZFFTI (n,w)
       call clock(t1)
       if(isw.eq.1)then
        call ZFFTF (n,cx,w)
       else
        call ZFFTB (n,cx,w)
       endif
       call clock(t2)
       tm=(t2-t1)
       perf(isw)=5.d0*n*log(dble(n))/log(2.d0)/tm*1.d-9

     enddo
     write(6,*) n,perf(1),perf(-1)
    enddo
    end
```

FFTPACK のサブルーチン

図 4.3-37 1 次元複素数 FFT(FFTPACK)の評価コード

ここで性能測定プログラムに使う評価コードについて説明する. この評価コードでは, 必要なデータ領域を確保し, FFT の入力データおよびワーク領域の初期化を行って1次元複素数フーリエ変換のサブルーチンを呼び出している. 実行時間の計測は, フーリエ変換のサブルーチンを呼び出す前後で, タイマールーチン(CLOCK)を挟んで行っている.

つぎに性能の算出方法について説明する. FFT の代表的な計算方法である Cooley-Tukey のアルゴリズムからデータ数 n の1次元複素数 FFT の演算数を数え上げると, 次式で表わされる.

$$演算数 = 5n \times \log_2(n)$$

この Cooley-Tukey のアルゴリズムの演算数は基数が 2 の場合の結果であるが, 基数が 2 以外でも大きく違わないため, 演算数を求めるには良い近似値となる. この方法で FFT の演算数を求め, 実行時間で除する事で性能(GFLOPS 値)を算出している.

図 4.3-38 は, データ数を 2^{11} (=2,048) から 2^{26}(=67,108,864)まで 2 のべき乗で増加させた時の FFT の性能である. 図中, fwd は順変換を bwd は逆変換を意味している.

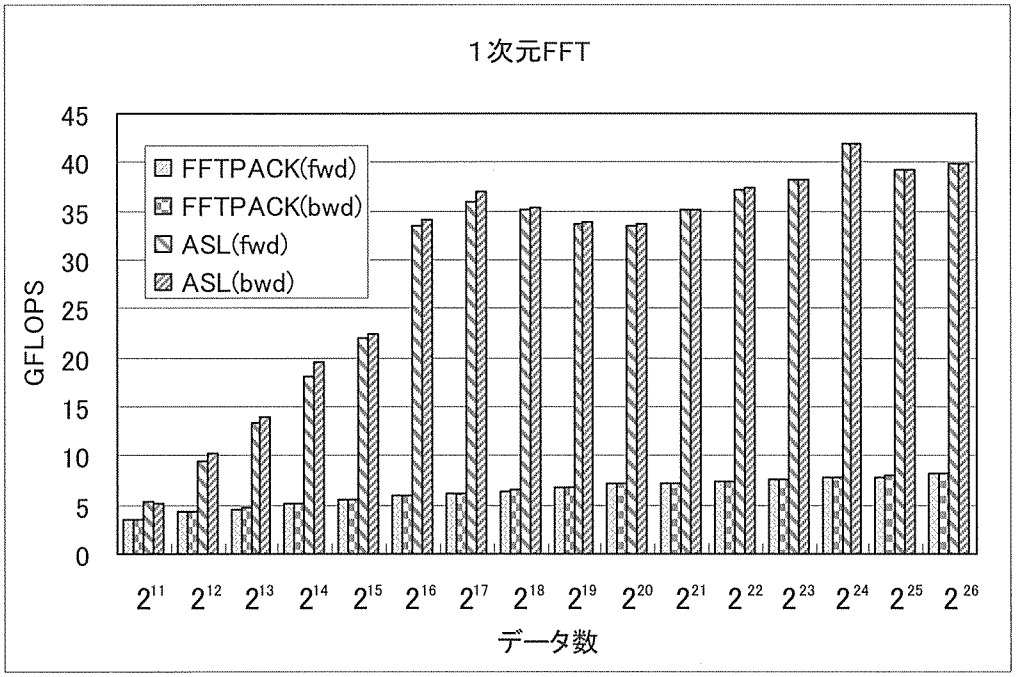

図 4.3-38 1 次元複素数 FFT の性能比較

FFTPACK は，データ数を増加しても大きな性能向上が得られない． ASL は，データ数を増加すると 2^{17}(=131,072)で 36GFLOPS と性能が向上し，最大性能は 2^{24}(=16,777,216)で 41GFLOPS となる．ASL と FFTPACK の性能を比較すると，データ数が大きいところで約 5 倍 ASL が高速である．

(3) ASL と FFTE の性能評価

ここでは2次元複素数フーリエ変換を ASL のサブルーチン ZFC2FB/ZFC2BF と，FFTE はベクトル用のサブルーチン ZFFT2D を用いて行っている(注 1)．それぞれの評価コードを，図 4.3-39 と図 4.3-40 に示す．これらのサブルーチンは，2次元の各次元のデータで離散フーリエ変換(順変換と逆変換)を行う．

```
! ２次元複素数 FFT (ASL) の評価コード
    integer, parameter :: ld=10241
    complex(kind=8) :: c(ld*ld),trigs(2*ld),wc(ld*ld)
    integer :: ifax(40)
    real(kind=8):: t1,t2,tm,perf(-1:1)
    do ii=1024,10240,1024
    do isw=1,-1,-2
      nx=ii
      ny=ii
      lx=nx+1
      ly=ny+1
      call init(c,lx,ly,nx,ny)

      call ZFC2FB(nx,ny,c,lx,ly,0,ifax,trigs,wc,ierr)
      call clock(t1)
      call ZFC2BF(nx,ny,c,lx,ly,isw,ifax,trigs,wc,ierr)
      call clock(t2)
      tm=(t2-t1)
      perf(isw)=5.d0*nx*ny*(log(dble(nx))+log(dble(ny)))/log(2.d0)/tm*1.d-9

     enddo
     write(6,*) ii,perf(1),perf(-1)
    enddo
    end
    subroutine init(c,lx,ly,nx,ny)
    complex(kind=8) :: c(lx,ly)
    do j=1,ny
     do i=1,nx
      c(i,j) = dcmplx(cos(sqrt(2.d0)*i),sin(sqrt(2.d0)*j))
     enddo
    enddo
    end
```

ASL の FFT サブルーチン

図 4.3-39 ２次元複素数 FFT(ASL)の評価コード

```
! ２次元複素数 FFT (FFTE) の評価コード
    integer, parameter :: ld=10240
    complex(kind=8) :: a(ld*ld)
    real(kind=8) :: t1,t2,tm,perf(-1:1)
    do ii=1024,10240,1024
     if(mod(ii,7).eq.0) cycle
     do isw=1,-1,-2
      nx=ii
      ny=ii
      call init(a,nx,nx,nx,ny)

      call ZFFT2D(a,nx,ny,0)
      call clock(t1)
      call ZFFT2D(a,nx,ny,-isw)
      call clock(t2)
      tm=(t2-t1)
      perf(isw)=5.d0*nx*ny*(log(dble(nx))+log(dble(ny)))/log(2.d0)/tm*1.d-9

     enddo
     write(6,*) ii,perf(1),perf(-1)
    enddo
    end
    subroutine init(c,lx,ly,nx,ny)
    complex(kind=8) :: c(lx,ly)
    do j=1,ny
     do i=1,nx
      c(i,j) = dcmplx(cos(sqrt(2.d0)*i),sin(sqrt(2.d0)*j))
     enddo
    enddo
    end
```

FFTE のサブルーチン

図 4.3-40 ２次元複素数 FFT(FFTE)の評価コード

図4.3-41は2次元データの各次元とも1,024の増分で1,024×1,024から10,240×10,240までデータ数を増加させた時のFFTの性能である. ただし, FFTEはデータ数に2, 3, 5以外の約数が入ると計算できないため, 約数に7が入るデータ数は除外している(注 2). 実行時間の計測は, (2)と同様, フーリエ変換のサブルーチンを呼び出す前後で行い, タイマールーチンを呼ぶ前にFFTの入力データとワーク領域の初期化を行っている. 演算数は1次元のFFTでの算出方法を2次元のFFTに拡張し, 性能を求めている.

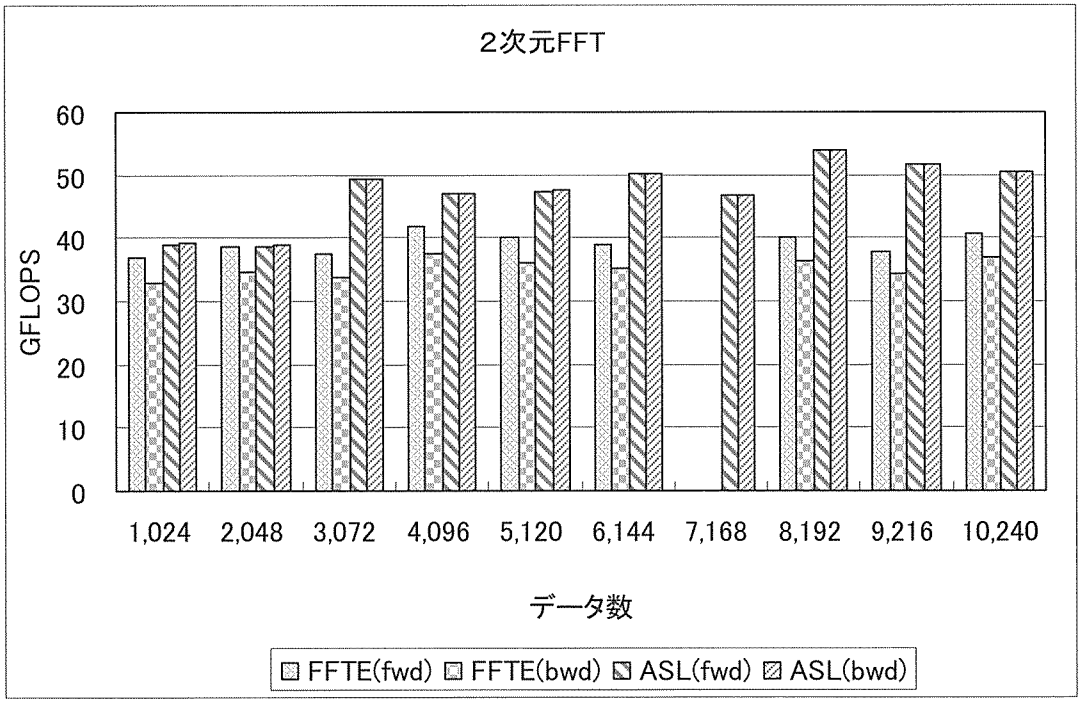

図 4.3-41 2 次元複素数 FFT の性能比較

FFTEはデータ数により32～42GFLOPSの性能である. ASLはデータ数が8,192×8,192のとき, 最大性能の54GFLOPSとなる. ASLとFFTEの性能を比較すると1.1～1.4倍ASLが高速である.

(注 1) FFTEは, ベクトル用およびスカラ用に最適化されたFFTのサブルーチンが用意されている. それぞれのプラットフォームに適しているサブルーチンを利用すること.

(注 2) FFTEは, データ数に2,3,5以外の約数が入っているとFFTの計算ができない仕様である. 例えば7,168は約数に7が入り計算できない. 一方ASLのFFTは, データ数に約数の制約はなく任意のデータ数でFFTの計算ができる.

(4) 性能評価のまとめ

科学技術計算ライブラリASLのFFTサブルーチンとフリーソフトウェアのFFTライブラリであるFFTPACK, FFTEとの間で性能を比較している. その結果, ASLのFFTサブルーチンが高い性能を示している. FFTEにはベクトル用に最適化されているFFTサブルーチンが用意されており, 実行効率が30%～40%と比較的よい性能が得られているが, ASLのFFTサブルーチンは実行効率が最大で50%を超え, より高速である.

SXでFFTの計算を行う場合は, SXのハードウェア向けに高度な最適化が施されている科学技術計

算ライブラリASLを使うことで，フリーソフトウェアのFFTライブラリを上回る性能が引き出せることがわかる．

4.3.7 リストベクトルを含むDOループの最適化

(1) リストベクトルの概要

リストベクトルとは，図4.3-42に示すように間接参照される配列IXにしたがって，配列Aの参照または更新をベクトル処理で行うものである．図4.3-43に図4.3-42に示したコードのメモリアクセスパターンを示す．リストベクトルを用いることで，連続でないランダムなメモリアクセスをベクトル処理として行うことができる．リストベクトルはDOループ中のIF文の除去や，ベクトル化できないデータ参照関係を回避する平面法などに活用されている．平面法については，「高速化活動報告第4号」を参照のこと．

```
!参照
do  i=1,n
  T(i)=A(IX(i))
enddo

!更新
do  i=1,n
  A(IX(i))=T(i)
enddo
```

図 4.3-42 リストベクトルのコード

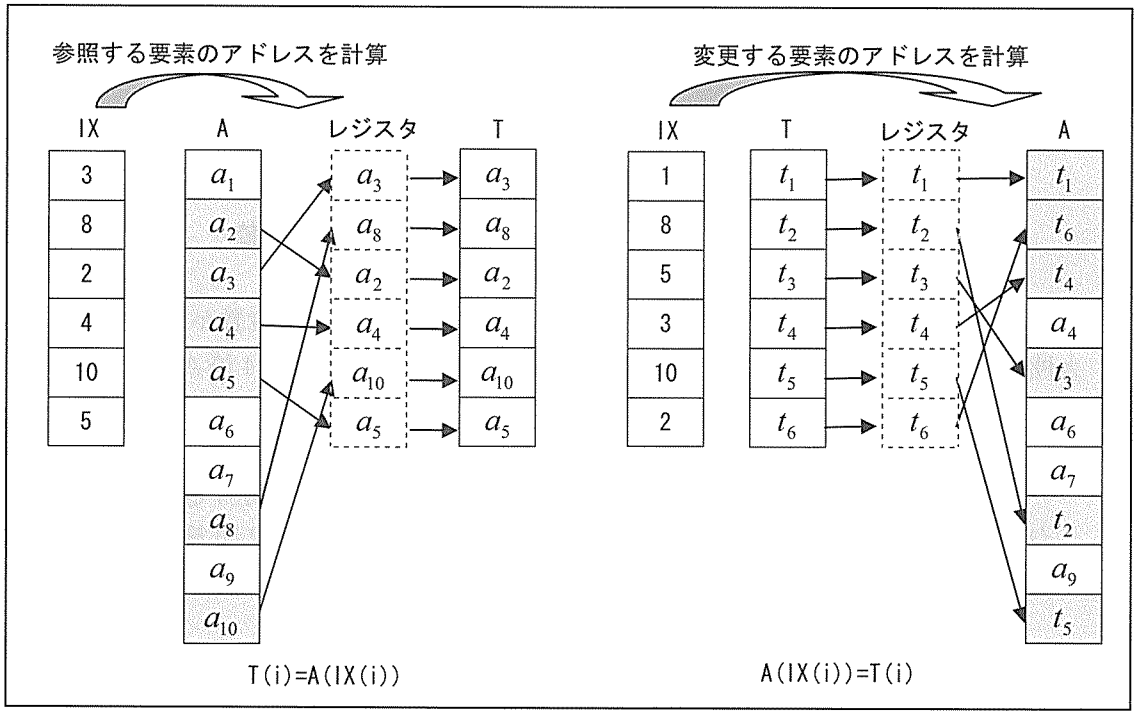

図 4.3-43 リストベクトルのメモリアクセス

(2) 事例1:リスト参照のメモリレイテンシの隠ぺい

リストベクトルによる参照や更新が同一 DO ループ内に複数回現れるプログラムの高速化の手法について説明する．

図4.3-44にリストベクトルによる参照や更新が同一DOループ内に複数回現れるコード(注)を示す．

間接参照 n2 による配列 p の参照は，間接参照 n1 による同配列 p への更新の完了を待つためメモリレイテンシ(遅延)が表面化し，演算性能を発揮できない．そこで配列 p のリストベクトルによる参照と更新のタイミングをずらすため，一度別の作業配列に連続データとして保存する．

```
121: +------>       do icol=1,ncol
122: |              !cdir nodep
123: |V----->         do edge=esep(icol),esep(icol+1)-1
124: ||                 n1=e2n(edge,1)
125: ||                 n2=e2n(edge,2)
 :
141: ||                 x1=u(n1,1)
142: ||                 y1=u(n1,2)
143: ||                 x2=u(n2,1)
144: ||                 y2=u(n2,2)
 :
161: ||                 !userfunc はインライン化によりベクトル化されるユーザ定義関数
162: ||                 v=userfunc(x1,y1,z1,x2)
 :
191: ||                 p(n1)=p(n1)+v
192: ||                 p(n2)=p(n2)-v
 :
209: |V-----          enddo
210: +-------       enddo
```

図 4.3-44 リストベクトルが同一 DO ループに複数回現れるコード

図 4.3-45 に作業配列を使って最適化するコードを示す．先ず配列 p を間接参照 n1,n2 により参照し，それぞれの作業配列 pn1,pn2 に連続データとして保存する．次に作業配列 pn1,pn2 を使って連続アクセスのデータで演算する．最後に演算結果を，作業配列から元の配列 p へ書き戻している．このようにすることで，配列 p の間接参照による参照を更新の前にまとめて行い，更新完了の待ちによる間接参照時のメモリレイテンシの影響を少なくできる．

```
121: +------>        do icol=1,ncol
122: |              ! リストベクトルによる参照処理の集約
123: |V----->          do edge=egsep(icol),egsep(icol+1)-1
124: ||                 n1=e2n(edge,1)
125: ||                 n2=e2n(edge,2)
 :
141: ||                 x1_a(edge)=u(n1,1)
142: ||                 y1_a(edge)=u(n1,2)
143: ||                 x2_a(edge)=u(n2,1)
144: ||                 y2_a(edge)=u(n2,2)
 :
161: ||                 pn1(edge)=p(n1)
162: ||                 pn2(edge)=p(n2)
 :
179: |V-----          enddo
180: |              ! 連続ベクトルによる演算
181: |V----->          do edge=egsep(icol),egsep(icol+1)-1
182: ||                 v=userfunc(x1_a(edge),y1_a(edge),x2_a(edge),y2_a(edge))
 :
211: ||                 pn1(edge)=pn1(edge)+v
212: ||                 pn2(edge)=pn2(edge)-v
 :
229: |V-----          enddo
230: |              ! リストベクトルによる更新処理の集約
231: |              !cdir nodep
232: |V----->          do edge=egsep(icol),egsep(icol+1)-1
233: ||                 n1=e2n(edge,1)
234: ||                 n2=e2n(edge,2)
 :
250: ||                 p(n1)=pn1(edge)
251: ||                 p(n2)=pn2(edge)
 :
268: |V-----          enddo
269: +------        enddo
```

図 4.3-45 作業配列を使った演算コード

チューニング前後のコードを使って，SX-9 1CPU で性能評価を行う．図 4.3-46 に FTRACE の情報を示す．org がチューニング前のコード，tune がチューニング後のコードを表している．配列 p の間接参照を更新より先に一括して行っていることで，メモリレイテンシを隠ぺいでき，図 4.3-46 で示されるようにメモリネットワーク競合時間(NETWORK)が半減し，CPU 時間比にして 1.95 倍に性能が向上している．

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
org	100	24.849(12.8)	248.493	6508.8	1817.5	99.95	250.1	24.843	0.001	0.001	2.039	18.972
tune	100	12.707(6.5)	127.074	16339.5	3554.2	99.80	250.1	12.640	0.016	0.012	2.301	8.792

図 4.3-46 チューニング前後の FTRACE 情報

(注)本コードは，間接参照 n1, n2 がそれぞれ最内の DO ループの中で重複する値を取らないことと，間接参照 n1 と n2 の間にも値の重複がないことを想定している．いずれも重複がある場合，ベクトル化によりデータの不整合が起こる．コンパイラはデータ依存性を判断できないため，図 4.3-44 の 122 行目および図 4.3-45 の 231 行目に指示行「!cdir nodep」を追加し，コンパイラに重複がないことを指示することでベクトル化を行っている．

(3) 事例2:配列の次元の入れ替えによるリストベクトルの回避

リストベクトル処理はメモリアクセスのレイテンシを隠ぺいすることが難しく，メモリを連続にアクセスする処理と比べて，処理時間を多く必要とし実効性能が低下する．

図 4.3-47 にリストベクトルのコードを示す．最内の i の DO ループの配列 ain はリスト il2, il1, ir1, ir2 により間接参照され，リストベクトルとなっている．

```
68: +------>            do k=1,kmax
69: |+----->              do j=1,jmax
70: ||V---->                do i=2,imax
71: |||                       il2=ilir(i,1)
72: |||                       il1=ilir(i,2)
73: |||                       ir1=ilir(i,3)
74: |||                       ir2=ilir(i,4)
75: |||                       stbc=wstbc(i)
76: |||                       dqr0=ain(i,j,k)
77: |||                       dqr1=ain(ir1,j,k)
78: |||                       dqr2=ain(ir2,j,k)
79: |||                       dql0=ain(i-1,j,k)
80: |||                       dql1=ain(il1,j,k)
81: |||                       dql2=ain(il2,j,k)
```

図 4.3-47 リストベクトル処理のコード

配列 ain に対して，i,j,k の次元から j,k,i の次元へ入れ替えを行うとともに，i の DO ループを最外に移動させる．図 4.3-48 に配列の次元を入れ替えたコードを示す．この変更により，配列 ain の参照は j,k の DO ループに対し連続ベクトルとなり，リストベクトルが回避される．加えてこの変更により，添え字 j が配列の下限から上限までの値を取ることで j,k の DO ループが一重化でき，ベクトル長が長くなる（注）．

```
68: V----->            do i=2,imax
69: |                    il2=ilir(i,1)
70: |                    il1=ilir(i,2)
71: |                    ir1=ilir(i,3)
72: |                    ir2=ilir(i,4)
73: |                    stbc=wstbc(i)
74: |W---->              do k=1,kmax
75: ||*--->                do j=1,jmax
76: |||                      dqr0=ain(j,k,i)
77: |||                      dqr1=ain(j,k,ir1)
78: |||                      dqr2=ain(j,k,ir2)
79: |||                      dql0=ain(j,k,i-1)
80: |||                      dql1=ain(j,k,il1)
81: |||                      dql2=ain(j,k,il2)
```

図 4.3-48 配列の次元を入れ替えたコード

チューニング前後のコードを使って，SX-9 1CPU で性能評価を行う．図 4.3-49 に FTRACE の情報を示す． org がリストベクトル処理主体のコード，tune が配列の次元を入れ替えたコードを表している．リストベクトルの回避とベクトル長が長くなることで，メモリネットワーク競合時間が大幅に削減され，CPU 時間比にして 11.41 倍に性能が向上している．

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
org	1	33.964(30.1)	33963.691	7920.3	1748.9	99.55	99.0	33.964	0.000	0.000	1.875	29.967
tune	1	2.977(2.6)	2976.662	46796.1	19955.2	99.50	250.0	2.976	0.000	0.000	0.010	1.418

図 4.3-49 チューニング前後の FTRACE 情報

(注)図 4.3-48 のコードにおける j,k の DO ループは，ループ長 jmax×kmax の一重ループとして処理される．例えば ain(j,k,i)への参照は，1～jmax×kmax の値を取るループ変数 jk を用いることで ain(jk,1,i)への参照と同様に処理される．

(4) 事例3:IF 文の変更によるリストベクトルの回避

図 4.3-50 のコードは，22 行目の IF 文で決定される変数 j を用いて配列 c を間接参照しているため，リストベクトルとなる(注)．この場合，IF 文の処理内容を変更することでリストベクトルを回避できる．

```
21: V------>        do i=1,n
22: |                 if ( x(i) .ge. 0.0 ) then
23: |                   j=2
24: |                 else
25: |                   j=1
26: |                 endif
27: |                 z1(i)=z1(i)+y1(i)*c(1,j)
28: |                 z2(i)=z2(i)+y2(i)*c(2,j)
29: V-------        enddo
```

図 4.3-50 IF 文で決定した変数を用いて間接参照を行うコード

図 4.3-51 にリストベクトルの回避コードを示す．22 行目以降の IF 文を配列 c の値を直接参照し，新たに導入した作業変数 c1,c2 に格納する．これらの作業変数 c1,c2 を用いた 29 行目以降の演算は連続ベクトルとなるため，リストベクトルが回避される．加えて，IF 文の処理は配列 c の値を定数としてレジスタ上で利用できるため，メモリアクセスも大幅に低減される．

```
21: V------>        do i=1,n
22: |                 if ( x(i) .ge. 0.0 ) then
23: |                   c1=c(1,2)
24: |                   c2=c(2,2)
25: |                 else
26: |                   c1=c(1,1)
27: |                   c2=c(2,1)
28: |                 endif
29: |                 z1(i)=z1(i)+y1(i)*c1
30: |                 z2(i)=z2(i)+y2(i)*c2
31: V-------        enddo
```

図 4.3-51 IF 文内で配列を直接参照するコード

チューニング前後のコードを使って，SX-9 1CPU で性能評価を行う．図 4.3-52 に FTRACE の情報を示す．org がリストベクトルのコード，tune がリストベクトル回避コードを表している．リストベクトルの回避とメモリアクセスの低減によりメモリネットワーク競合時間が大幅に削減され，CPU 時間比にして 9.90 倍に性能が向上している．

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
org	1	22.664(27.6)	22663.970	8405.8	1764.9	99.73	255.8	22.664	0.000	0.000	1.275	20.002
tune	1	2.287(2.8)	2286.695	70192.4	17492.5	99.68	255.8	2.287	0.000	0.000	0.023	0.693

図 4.3-52 チューニング前後の FTRACE 情報

(注)ループ変数 i の値それぞれに応じた変数 j の値は，ベクトルレジスタもしくはコンパイラが自動生成する作業配列に，一度ベクトルデータとして格納される．そのため j による配列 c の間接参照は，この格納されたベクトルデータを元にリストベクトルとして処理される．

4.3.8 SOR 法のベクトル化(Red Black 法)

(1) SOR 法の概要

SOR 法(Successive Over-Relaxation method)とは, n 元連立方程式 Ax=b を反復法で解く手法の一つである. 図 4.3-53 に示すとおり, ある要素 p(i,j,k)の計算には近傍の要素 p(i-1,j,k), p(i+1,j,k), p(i,j-1,k), p(i,j+1,k), p(i,j,k-1), p(i,j,k+1)が必要な計算を行う.

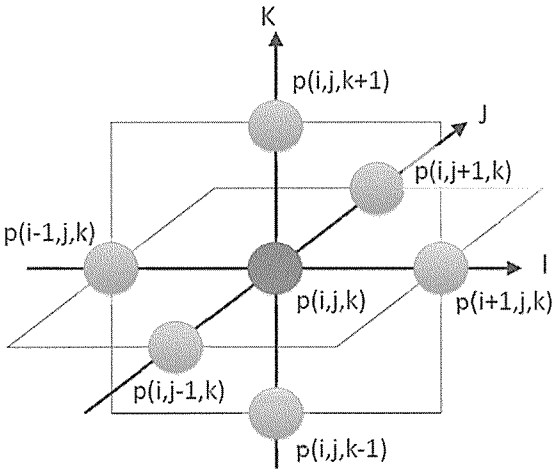

図 4.3-53 SOR 法計算の関係図

図 4.3-54 に SOR 法のコードを示す. p(i,j,k)の値を計算する際に, p(i-1,j,k)の要素を参照する必要がある. ベクトル化の対象となる最内の DO ループでは, p(i-1,j,k)の値は 1 回前で定義されており(注 1), この定義・参照関係がベクトル化を阻害する依存関係になる. そのためベクトル化できず(注 2), SX-9 で高い実効性能を得ることができない.

(注 1)p(i+1,j,k)は参照された後で値が更新されるので, ベクトル化を阻害する要因にはならない.

(注 2)図 4.3-54 の例のように, コンパイラはベクトル化可能な部分のみベクトル化を行う. これは部分ベクトル化と呼ばれ編集リストでは「S」で示される.

```
115  +------>          do k = n_bnd + 1, n_bnd + n_cell
116  |+----->           do j = n_bnd + 1, n_bnd + n_cell
117  ||V---->            do i = n_bnd + 1, n_bnd + n_cell
 :
141  |||    S              p_a = c0
142  |||    S       .      * ( p(i-1,j  ,k  ) * xm + p(i+1,j  ,k  ) * xp
143  |||    S       .        + p(i  ,j-1,k  ) * ym + p(i  ,j+1,k  ) * yp
144  |||    S       .        + p(i  ,j  ,k-1) * zm + p(i  ,j  ,k+1) * zp
145  |||    S       .        - rhs * ds * ds )
 :
148  |||    S              p(i,j,k) = p(i,j,k) + cnst_p * ( p_a - p0 )
149  ||V---               end do
150  |+-----             end do
151  +------            end do
```

図 4.3-54 SOR 法のコード

(2) Red Black 法

定義・参照の依存関係が存在することでベクトル化ができない DO ループに対して, 依存関係が生じない順序で配列要素をアクセスすることによりベクトル化を可能にする手法の一つに Red Black 法が

ある. 図 4.3-55 に 3 次元の Red Black 法の概念図を示す. 配列要素を赤・黒と順番に色付けを行い, 赤の要素の計算と黒の要素の計算を行うループを分けることにより, ベクトル化を阻害する依存関係を排除できる.

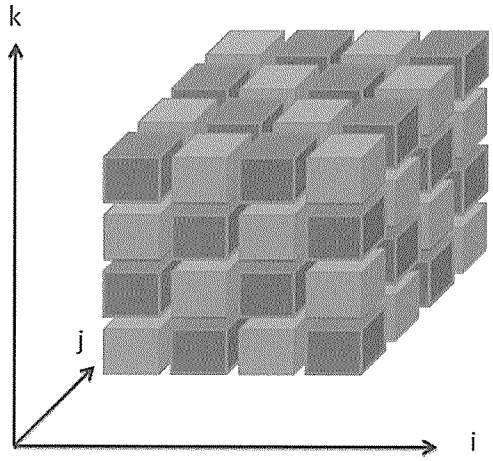

図 4.3-55 Red Black 法の概念図

3 次元の場合, 配列のアクセスは以下のように行われる(各次元のサイズが 8 の場合).

赤のループ:p(1,1,1),p(3,1,1),…,p(7,1,1),p(2,2,1),p(4,2,1),….

黒のループ:p(2,1,1),p(4,1,1),…,p(8,1,1),p(1,2,1),p(3,2,1),….

図 4.3-56 に Red Black 法のコードを示す. 最内 DO ループは上記に示した配列のアクセスのとおり増分 2 で繰り返される. 先頭要素から始まるループが赤のループ, 後続の先頭要素+1 から始まるループが黒のループに該当する. j, k の DO ループも増分 2 のアクセスになるように IF 文で処理を分岐している.

```
113 +-----> do k = n_bnd + 1, n_bnd + n_cell
114 |+-----> do j = n_bnd + 1, n_bnd + n_cell
115 ||        if(mod(k,2).eq.0.and.mod(j,2).eq.0.or.
116 ||   +      mod(k,2).eq.1.and.mod(j,2).eq.1) then
117 ||V---->   do i = n_bnd + 1, n_bnd + n_cell,2
  :
133 |||          p_a = c0
134 |||   .      * ( p(i-1,j  ,k  ) * xm + p(i+1,j  ,k  ) * xp
135 |||   .        + p(i  ,j-1,k  ) * ym + p(i  ,j+1,k  ) * yp
136 |||   .        + p(i  ,j  ,k-1) * zm + p(i  ,j  ,k+1) * zp
137 |||   .        - rhs * ds * ds )
138 |||          p(i,j,k) = p(i,j,k) + cnst_p * ( p_a - p0 )
139 ||V---     end do
140 ||       else if(mod(k,2).eq.1.and.mod(j,2).eq.0.or.
141 ||   +      mod(k,2).eq.0.and.mod(j,2).eq.1) then
142 ||V---->   do i = n_bnd + 2, n_bnd + n_cell,2
  :
```

赤のループの実行

黒のループの実行

図 4.3-56 Red Black 法のコード

(3) DO ループの一重化

図 4.3-56 に示したコードにおいて, 最内の DO ループがベクトル化の対象になるため, ベクトル長は

i の DO ループの半分のループ長になる(増分 2 でアクセスする)ため，十分なベクトル長が得られない．そこで，2 つのループを含めて 3 重ループを一重化することで，ベクトル長の拡大を図る．この例では，各次元に計算を行わない袖要素を含んでいるため，単純に DO ループの一重化ができない．そこで予め計算を行う要素の場合を「1」，行わない要素の場合を「0」とするマスクテーブルを用意しておき，IF 文の判定で真になる場合のみ計算を実行するようにする．図 4.3-57 に DO ループの一重化を行うコードを示す．コンパイラが配列 p の依存関係を解析できないので，コンパイラ指示行 NODEP を指定して，配列 p のアクセスに重なりがないことを明示する．

```
277 +-----> do icolor = 1, 2
278 |     !cdir nodep(p)
279 |V----> do ijk=icolor,max_cell*max_cell*max_cell,2
280 ||       if(matbl(ijk,1,1,icolor).eq.1.or.
281 ||    +     matbl(ijk,1,1,icolor+2).eq.1 ) then
 :
297 ||         p_a = c0
298 ||    .    * ( p(ijk-1,                1,1) * xm + p(ijk+1,                1,1) * xp
299 ||    .      + p(ijk-max_cell,         1,1) * ym + p(ijk+max_cell,         1,1) * yp
300 ||    .      + p(ijk-max_cell*max_cell,1,1) * zm + p(ijk+max_cell*max_cell,1,1) * zp
301 ||    .      - rhs * ds * ds )
 :
304 ||         p(ijk,1,1) = p(ijk,1,1) + cnst_p * ( p_a - p0 )
305 ||       endif
306 |V----   end do
307 +-----  end do
```

マスクテーブル配列 matbl の値が真の場合，ループの処理が実行される

図 4.3-57 マスクテーブルを使用するコード

(4) ADB の利用

SOR 法の計算は，4.3.8(1)で示しているように，ある要素の計算に近傍の 6 つの要素を使用するステンシル計算である．各要素は異なる繰り返しにおける計算で再び参照される．ON_ADB 指示行で，これらの要素を格納する配列 p を ADB に乗せることを指示する．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，メモリのアクセス回数を削減することができ，処理効率の向上が期待される．図 4.3-58 に ON_ADB 指示行を指定したコードを示す．

```
277 +-----> do icolor = 1, 2
278 |     !cdir nodep(p)
279 |     !cdir on_adb(p)
280 |V----> do ijk=icolor,max_cell*max_cell*max_cell,2
281 ||       if(matbl(ijk,1,1,icolor).eq.1.or.
282 ||    +     matbl(ijk,1,1,icolor+2).eq.1 ) then
 :
298 ||         p_a = c0
299 ||    .    * ( p(ijk-1,                1,1) * xm + p(ijk+1,                1,1) * xp
300 ||    .      + p(ijk-max_cell,         1,1) * ym + p(ijk+max_cell,         1,1) * yp
301 ||    .      + p(ijk-max_cell*max_cell,1,1) * zm + p(ijk+max_cell*max_cell,1,1) * zp
302 ||    .      - rhs * ds * ds )
 :
305 ||         p(ijk,1,1) = p(ijk,1,1) + cnst_p * ( p_a - p0 )
306 ||       endif
307 |V----   end do
308 +-----  end do
```

図 4.3-58 ON_ADB 指示行の挿入

(5) 性能評価

SX-9 1CPU を用いて性能評価を行う．評価を行うプログラムは以下のとおりである．

① SOR 法を最適化行わず記述したプログラム

② Red Black 法を用いてベクトル化を行ったプログラム

③ ②に対してマスクテーブルを用いてループの一重化を行ったプログラム

④ ③に対して再利用性の高い配列 p を ADB に乗せることを指示したプログラム

表 4.3-3 に，各次元のループ長が 64 の場合の性能特性を示す．

表 4.3-3 性能特性

プログラム	ベクトル演算率(%)	平均ベクトル長	実効性能(MFLOPS)
①	51.86	64.0	189.5
②	90.11	54.4	970.1
③	99.59	255.9	8,247.3
④	99.59	255.9	18,940.6

①のプログラムは，ベクトル化できない DO ループ構造であるため，コンパイラは部分的にしかベクトル化を行わず，ベクトル演算率は 51.9%となり 200MFLOPS に満たない性能値となる．②の 3 次元ループのままの Red Black 法のプログラムでは，ベクトル化は行えるものの，平均ベクトル長が短いことと，ベクトルループを外側で何度も繰り返すため，高い実効性能が得られない．③の 3 次元ループに対してマスクテーブルを用いて一重化したプログラムでは，平均ベクトル長が 255.9 となり，8.3GFLOPS の性能値となる．さらに④の ADB に再利用性の高い配列要素を乗せるプログラムでは，メモリへのアクセス回数を削減することができ，③と比較して 2 倍近い性能向上が得られる．

図 4.3-59 に，それぞれのプログラムで次元サイズを変更した場合の性能グラフを示す．すべてのケースにおいて，④が最も高い性能であることが分かる．ベクトル型プロセッサの SX-9 では高いベクトル演算率，十分な平均ベクトル長だけでなく，メモリ負荷を軽減する最適化が有効であることが示される．特に再利用性の高い要素を ADB に載せることで実効性能の向上が期待される．

Red Black 法は，元の SOR 法と計算順序が異なるため，結果に差異が生じる．そのため SOR 法のような反復計算において，収束までの繰り返し回数が元の SOR 法の計算より増える場合がある．極端に収束回数が増加する場合は，Red Black 法による高速化の効果が相殺される可能性があるので注意されたい．

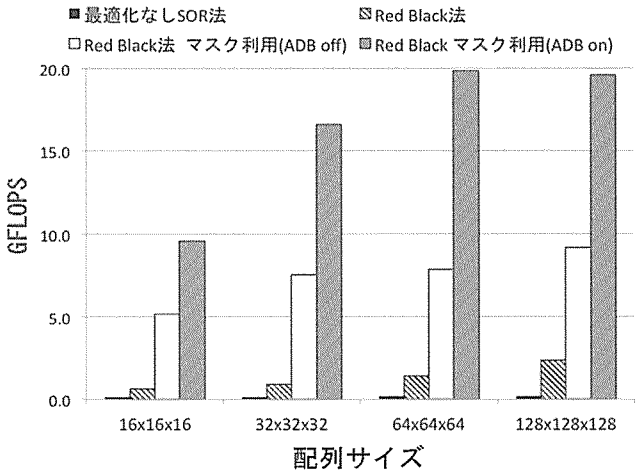

図 4.3-59 配列サイズと性能値

5 並列コンピュータ Express5800 の高速化

スーパーコンピューティング研究部　江川隆輔　岡部公起
情報部情報基盤課　伊藤英一　小野敏　山下毅
日本電気株式会社　撫佐昭裕　神山典　小久保達信
吉村健二　坂本英顕　金野浩伸
NEC システムテクノロジー株式会社　曽我隆

5.1 Express5800 の特徴

Express5800/A1080a-D（以下，Express5800）は，ノードあたり Intel Xeon X7560 プロセッサ（以下，X7560）を4台搭載し，512GB の主記憶装置を有するスカラ型コンピュータである．東北大学サイバーサイエンスセンターは，Express5800 を 6 ノード導入している．表 5.1-1 に Express5800 の主要諸元を示す．

表 5.1-1 Express5800 主要諸元

項目		諸元
最大演算性能		289.92GFLOPS
CPU 数		4
コア数		32
CPU	名称	Intel Xeon X7560
	動作周波数	2.26GHz
	最大演算性能	9.06GFLOPS（コアあたり） 72.48GFLOPS（CPU あたり）
	QPI	25.6GB/s
	キャッシュ	L1：コアあたり 32KB(命令)+32KB(データ)
		L2：コアあたり 256KB(命令/データ共用)
		L3：CPU あたり 24MB
主記憶装置	容量	512GB
	最大データ転送能力	34.1GB/s（CPU あたり）

X7560 は 8 コアを有する Intel 社製の 64bit プロセッサであり，メモリコントローラを CPU に内蔵し，CPU－主記憶装置間を直結している．また，QPI（quick path interconnect）アーキテクチャを用い，CPU 間の転送性能は，25.6GB/s である．図 5.1-1 に Express5800 のプロセッサ構成を示す．

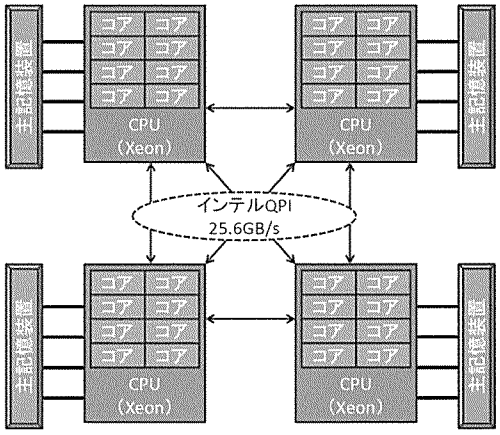

図 5.1-1 Express5800 のプロセッサ構成図

5.2 高速化技法

Express5800 で高い性能を得るためには，キャッシュの適切な利用とベクトル命令の活用が重要になる．ここでは Express5800 が搭載する X7560 の特性を活かした最適化手法を説明する．

5.2.1 キャッシュ・ブロッキングによる高速化

X7560 と主記憶装置間のメモリバンド幅は，34.1GB/s である．プロセッサあたりの理論ピーク性能が 72.48GFLOPS であることから，演算性能あたりのメモリバンド幅(Byte/FLOP)は 0.46 となり，SX-9 の 2.5 と比較してデータ供給能力が低いことがわかる．そのため，Express5800 ではキャッシュを有効に用いて，データ供給能力を補う工夫が必要になる．本節では，キャッシュ・ブロッキングによりキャッシュ内のデータを利用し，メモリのアクセス回数を削減する最適化手法について説明する．

(1) X7560 のキャッシュ構造

X7560 は，図 5.2-1 に示すように主記憶装置との間に L1 キャッシュ，L2 キャッシュ，L3 キャッシュの 3 階層のキャッシュを有している．L1 キャッシュと L2 キャッシュは各コアが個別に有しているが，L3 キャッシュは 8 つのコアで共有している．

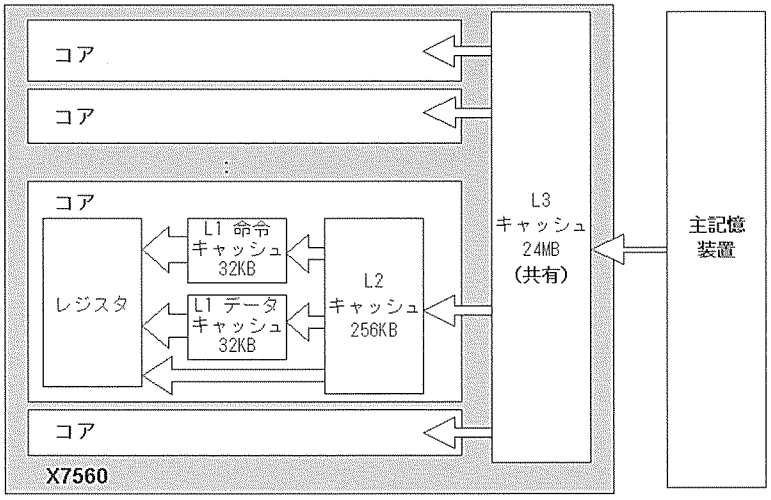

図 5.2-1 X7560 のキャッシュ構成図

キャッシュの特性として，L1，L2，L3 キャッシュの順にレジスタへ高速なデータ転送が可能である．したがって，プログラムの実効性能を向上させるためには，L1 キャッシュおよび L2 キャッシュを効果的に用いることが必要である．図 5.2-2 にキャッシュとプログラムの実効性能の関係を示す．この図は c=c+a(i,j)*b(i,j)の計算を行うループに対して，配列 a と配列 b の合計のメモリサイズを横軸に，実効性能を縦軸に取ったグラフである．配列 a と配列 b のデータが L1 キャッシュに乗っている間は高い性能を示しているが，データのサイズが L2 キャッシュの大きさになると性能は低下し，L3 キャッシュの大きさになるとさらに低下する．この結果から，高い実効性能を得るためには配列 a と配列 b のデータを L1，L2 キャッシュに乗せることが重要であることがわかる．

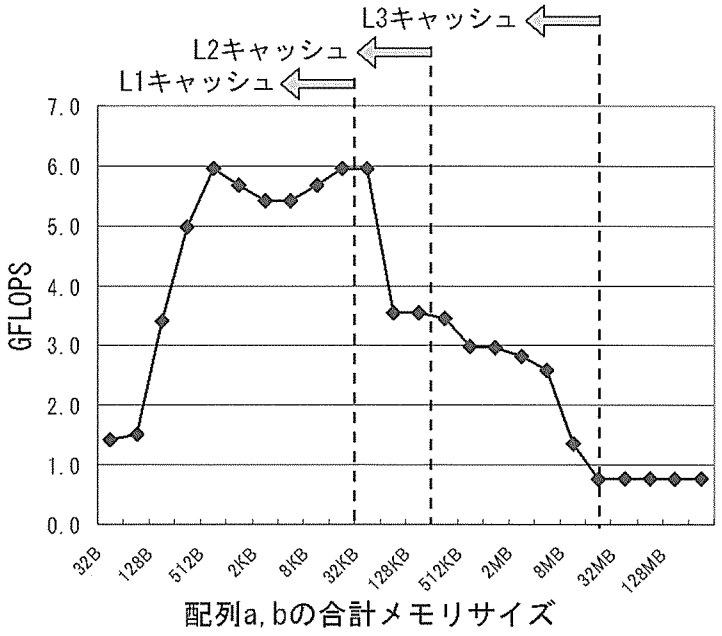

図 5.2-2 c=c+a(i,j)*b(i,j)の実効性能

(2) キャッシュ・ブロッキング

キャッシュ・ブロッキングとは，キャッシュ上にあるデータの再利用性を高めるため，DO ループをキャッシュサイズに合わせて分割する手法である．

図 5.2-3 に行列積を行うループのコードを示す．この DO ループは配列 a と配列 b の積を配列 c に集計する 3 重ループ構造である．最内の i の DO ループで，配列 a と配列 c の要素に連続アクセスしている．配列 b は i の DO ループに依存していないので，この最内側ループが実行されている間，配列 b は同じ要素が利用される．以下にキャッシュ・ブロッキングにおけるループ長の決め方と手順を示す．

配列 a は j の DO ループに依存していないので，j の DO ループを実行している間，一次元目の要素数(n=1,000 の場合，倍精度実数で 8,000 バイト)だけアクセスする．このサイズであれば配列 a は L2 キャッシュ上に乗せることができる．しかし，配列 c の全データがアクセスされるため，配列 a のデータは L2 キャッシュから追い出される．

```
parameter(n=1000)
   :
do k=1,n
  do j=1,n
    do i=1,n
     c(i,j) = c(i,j) + a(i,k)*b(k,j)
    enddo
  enddo
enddo
```

図 5.2-3 オリジナルの行列積ループ

ここで k の DO ループを実行している間, 配列 a のデータを L2 キャッシュに留め置くために, j の DO ループを分割し, 配列 c がアクセスする範囲を制限する. L2 キャッシュのサイズは 256KB であり, 32,768 個の倍精度実数データを乗せられる. 最内の i のループ長が 1,000 の場合, j のループの長さを 25 にすれば, 配列 a と配列 c の両方でアクセスする要素数は 26,000 個となり L2 キャッシュに乗せることができる. そこで図 5.2-4 に示すように, j と k の DO ループの外に増分 25 のループを設け, j の DO ループをループ長 25 でブロッキングする.

```
parameter(n=1000)
   :
do j2=1,n,25                 }  j の DO ループでキャッシュ・ブロッキング
  do k=1,n                   }
    do j=j2,j2+24            }
      do i=1,n
       c(i,j) = c(i,j) + a(i,k)*b(k,j)
      enddo
    enddo
  enddo
enddo
```

図 5.2-4 j の DO ループでブロッキングする行列積ループ

(3) 性能評価

表 5.2-1 にキャッシュ・ブロッキング前後の性能結果を示す. キャッシュ・ブロッキングの結果, メモリアクセス回数が削減され, 1.22 倍の性能向上が得られている.

表 5.2-1 行列積ループにおけるキャッシュ・ブロッキングの効果

	実効性能(GFLOPS)
キャッシュ・ブロッキング前の行列積ループ	2.24
j の DO ループでキャッシュ・ブロッキング	2.74

5.2.2 ベクトル命令の利用

X7560 のコアは，1 クロックサイクルあたり最大 4 演算を実行可能なアーキテクチャを採用している．このアーキテクチャを最大限に活用するため，コンパイラによるベクトル化が重要となる．

(1) ベクトル化の利用方法

コンパイラがベクトル化を行うのは，最内の DO ループに対し，以下の条件を満たす場合である．

① データの定義・引用に依存関係がないこと

② ループ中に複数のデータ型の演算(倍精度実数と単精度実数など)が混在しないこと

③ 間接参照によるメモリアクセスがないこと

Express5800 用のコンパイラ f95 は，コンパイルオプションを指定しなくても，ベクトル化可能なループをベクトル化する．プログラム中の DO ループがベクトル化されることは，コンパイル時にオプション「-vec-report3」を指定することで確認できる．図 5.2-5 にベクトル化される例と，図 5.2-6 にベクトル化されない例を示す．

図 5.2-5 では，DO ループがベクトル化の必要要件を満たしているので，コンパイラはベクトル化を行う．しかしながら図 5.2-6 では，ループ中の配列 c が配列 itbl を用いた間接参照となるので，コンパイラがデータの依存関係を解析できないため，ベクトル化が行われない．

```
      subroutine sub(n,a,b,c)
      real*8 a(n,n),b(n,n),c(n,n)
       do j=1,n
         do i=1,n
           c(i,j) = c(i,j) + a(i,j)*b(i,j)
         enddo
       enddo
       return
       end

test.f(7): (col. 7) remark: loop was not vectorized: not inner loop.
test.f(6): (col. 9) remark: LOOP WAS VECTORIZED.
```

ベクトル化された

図 5.2-5 ベクトル化が行われる例

```
      subroutine sub(n,a,b,c,d)
      real*8 a(n,n),b(n,n),c(n,n)
      integer*4 itbl(n)
      do j=1,n
        do i=1,n
          c(itbl(i),j) = c(itbl(i),j) + a(i,j)*b(i,j)
        enddo
      enddo
      return
      end

test.f(8): (col. 7) remark: loop was not vectorized: not inner loop.
test.f(7): (col. 9) remark: loop was not vectorized: existence of vector dependence.
test.f(6): (col. 11) remark: vector dependence: assumed ANTI dependence between c line 6 and c line 6.
test.f(6): (col. 11) remark: vector dependence: assumed FLOW dependence between c line 6 and c line 6.
```

配列 c の依存関係が解析できないため
ベクトル化できない

図 5.2-6 ベクトル化が行われない例

(2) ベクトル化による性能

図 5.2-3 のコードを用いて，ベクトル化される場合と，ベクトル化を抑止する場合の性能差を示す．なおベクトル化を抑止するため，コンパイルオプション「-no-vec」を利用している．表 5.2-2 に示しすよ

うに，ベクトル化を行うことで 1.66 倍の性能差になることがわかる．

表 5.2-2 行列積がベクトル化された場合とベクトル化されない場合の実効性能

	実効性能(GFLOPS)
ベクトル化を抑止する場合	1.35
ベクトル化される場合	2.24

6 並列処理

スーパーコンピューティング研究部　江川隆輔　岡部公起
情報部情報基盤課　伊藤英一　小野敏　山下毅
日本電気株式会社　撫佐昭裕　神山典　小久保達信　金野浩伸
NEC システムテクノロジー株式会社　曽我隆　塩田和永

6.1 並列処理の概要

並列処理とは，並列に実行可能な処理を複数のプロセッサ(あるいはコア)を用いて実行することである．単一のプロセッサでのみ実行可能なプログラムを逐次処理プログラムと呼び，逐次処理プログラムを並列処理プログラムに書き換えることを並列化と言う．並列化を行うことにより，プログラムの実行時間を短縮することができる．

並列処理には，コンピュータアーキテクチャに応じた以下の方式がある．

(1) 共有メモリ型並列処理

共有メモリ型並列処理とは，図 6.1-1 のように複数のプロセッサが単一のメモリ空間を共有しながら並列処理を行うものである．これは，コンパイラの自動並列化機能やコンパイル指示行の挿入により並列化を行うものである．東北大学サイバーサイエンスセンターにおける SX-9 では 16 並列，Express5800 では 32 並列まで実行することができる．

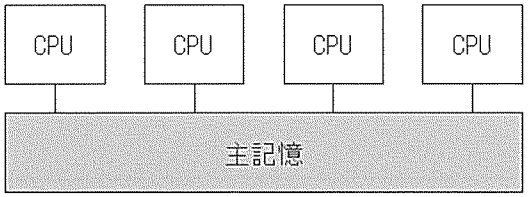

図 6.1-1 共有メモリマシンのアーキテクチャ

(2) 分散メモリ型並列処理

分散メモリ型並列処理とは，図 6.1-2 のようにネットワークを介して接続されている，複数の独立したメモリ空間を持つコンピュータで行われる並列処理のことである．この並列処理では，処理を分割し異なるプロセッサで別々に実行する．このとき，プロセッサ間で情報交換が必要な場合，MPI(Message Passing Interface)等の通信ライブラリを用いてデータのやり取りを行う．東北大学サイバーサイエンスセンターにおける SX-9, Express5800 では共に 64 並列まで実行することができる．

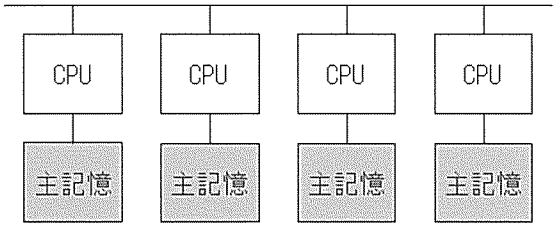

図 6.1-2 分散メモリマシンのアーキテクチャ

6.2 MPI プログラムの概要

6.2.1 MPI とは

MPI とは，MPI フォーラム(注 1)により開発・規格化された分散メモリ型並列処理におけるデータ通信のための標準規格であり，複数のプロセス(注 2)間でデータをやり取りするために用いるメッセージ通信操作のサブプログラム仕様の規格である．

主な特徴を以下に示す．

- FORTRAN, C から呼び出す通信操作のサブプログラムであり，ライブラリで提供される．
- 様々なコンピュータシステムの MPI 実装環境において同じソースコードで利用できる．
- 共有メモリ型並列処理に比べてプログラマの負担が大きいが，大規模計算が行える．

(注　1)MPI フォーラムの公式 Web ページは http://www.mpi-forum.org である．

(注　2)プロセス:コンピュータのプロセッサ上で自律的に動作するプログラムの実体

6.2.2 MPI のプログラム例

ここでは，総和計算を行う FORTRAN プログラムを例に MPI プログラムの説明を行う．

(1) 総和計算のプログラム

図 6.2-1 は，1 から 100 までの整数の和を求めるコードである．

```
program main
parameter(N=100,i1=1,i2=N)
isum = 0
do i = i1, i2
  isum = isum + i
enddo
write(6,*) "sum=",isum
stop
end program
```

図 6.2-1　総和計算のコード例

図 6.2-1 に示した総和計算の例では，1 から 100 まで順に和を求めている．この総和計算では，DO ループを複数に分割し，分割したループごとに部分和を取り，それぞれの部分和を集計して求めることができる．このように DO ループを分割した部分について，部分和を並列に実行することで並列処理することができる．

(2) 処理の分割

図 6.2-2 は，DO ループを均等に 4 分割し，4 プロセスで実行する場合の概念図である．各プロセスがそれぞれ DO ループのどの部分を担当するか，プロセス識別番号(0～3)で表している．MPI ではプロセスの識別番号のことを rank 番号と呼ぶ．各プロセスは担当する DO ループの部分和を求める計算を行う．

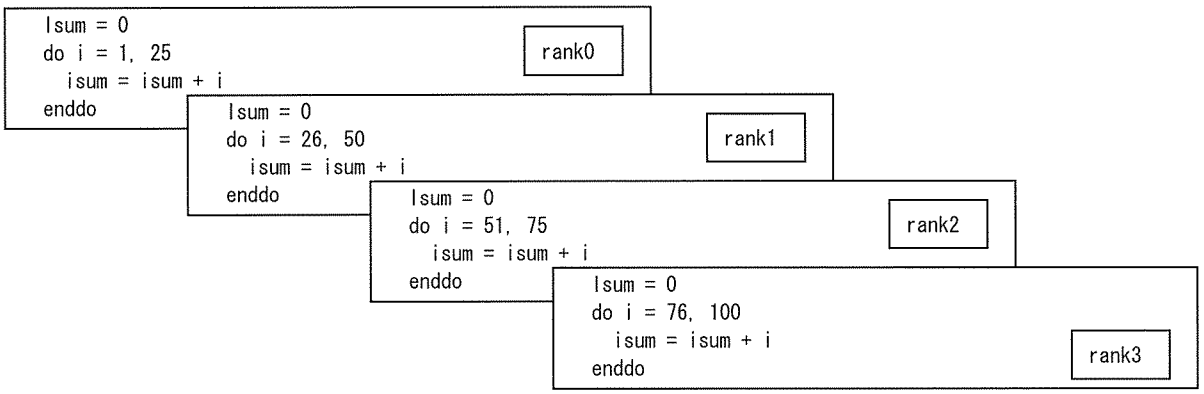

図 6.2-2 DO ループ分割の概念図

図 6.2-2 は並列数 4 の例であるが，プロセス数に応じて，各プロセスが分担する領域の始点と終点を下式①，②のように rank 番号から求める必要がある．ここで nprocs が全プロセス数を，myrank が各プロセスの rank 番号を表している．ただし，各プロセスでループを均等に分割できることを前提にしている．

① 始点　ist = ((i2-i1+1)/nprocs)*myrank+1
② 終点　ied = ((i2-i1+1)/nprocs)*(myrank+1)

この式によって求められる ist，ied は表 6.2-1 のとおりである．nprocs，myrank を使用することにより，図 6.2-3 のように各プロセスの処理を共通のコードで表すことができる．

表 6.2-1 rank 番号ごとの ist，ied の値

nprocs=4	ist	ied
myrank=0	1	25
myrank=1	26	50
myrank=2	51	75
myrank=3	76	100

```
ist = ((i2-i1+1)/nprocs)*myrank+1
ied = ((i2-i1+1)/nprocs)*(myrank+1)
isum = 0
do i = ist, ied
  isum = isum + i
enddo
```

図 6.2-3 部分和のコード例

(3) プロセス間通信

全体の総和は，各プロセスで求められた部分和を代表プロセスで集計して求める．集計には MPI による通信ライブラリを使用する．図 6.2-4 は，プロセス間の通信と部分和を集計する例であり，データの送受信を行う MPI の一対一通信サブルーチン(MPI_RECV，MPI_SEND)を用いている．rank 番号が 0 以外のプロセスは，それぞれの部分和を rank 番号 0 に送信する．rank 番号 0 のプロセスは，自プロセス以外のすべてのプロセスから送られる部分和を(nprocs-1)回受信し，受信した部分和 itmp を isum に集計する．

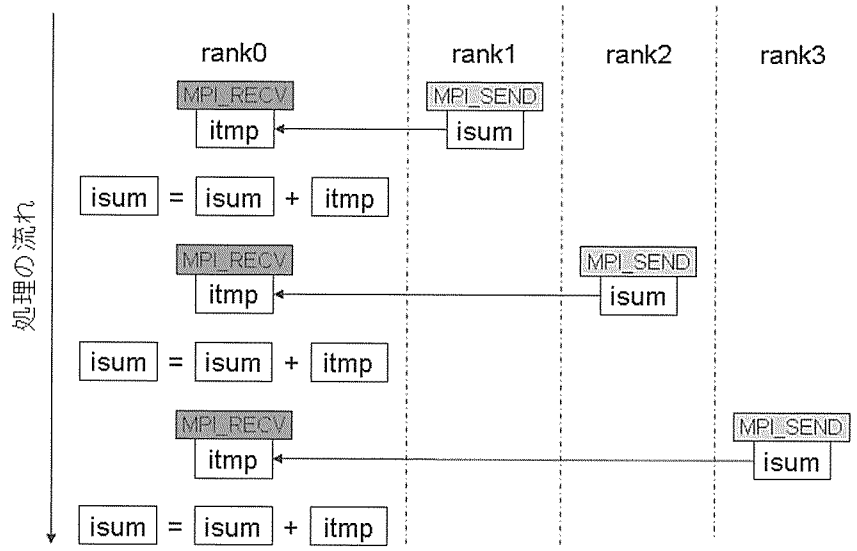

図 6.2-4 総和処理の流れ

(4) 総和計算の MPI コード例

以上を総合して，図 6.2-1 のプログラムを並列化したコードを図 6.2-5 に示す．

```
program example1
include 'mpif.h'                                                    !①
integer status(MPI_STATUS_SIZE)                                     !②
parameter(N=100, i1=1, i2=N)
integer nprocs, myrank, source, tag, ierr, ist, ied, i, isum, itmp
call MPI_INIT(ierr)                                                 !③
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)                    !④
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)                    !⑤

ist = ((i2-i1+1)/nprocs)*myrank+1
ied = ((i2-i1+1)/nprocs)*(myrank+1)
isum = 0
do i = ist, ied
  isum = isum + i
enddo

if(myrank .eq. 0) then
  do i = 1, nprocs-1
    source = i
    tag = i
    call MPI_RECV(itmp, 1, MPI_INTEGER, source, tag, MPI_COMM_WORLD, status, ierr)   !⑥
    isum = isum + itmp
  enddo
else
  tag = myrank
  call MPI_SEND(isum, 1, MPI_INTEGER, 0, tag, MPI_COMM_WORLD, ierr)                 !⑦
endif
if(myrank .eq. 0) write(6,*) "sum=", isum
call MPI_FINALIZE(ierr)                                             !⑧
stop
end
```

図 6.2-5 並列化コード例(総和計算)

この例で使用したMPIの宣言部分とサブルーチンについて以下に示す.

① mpif.h
MPIで予約されている定数が定義されている. MPIサブルーチンを利用するすべてのサブルーチン, 関数においてinclude文で引用する必要がある.

② status(MPI_STATUS_SIZE)
下記MPI_RECVの第七引数で使用されるメッセージ情報.

③ MPI_INIT
MPIの実行開始を宣言するサブルーチンであり, すべてのMPIサブルーチンに先立ち, 呼び出される必要がある. 引数(ierr)に実行結果の状態を受け取る(以下同様).

④ MPI_COMM_SIZE
MPI のプロセス管理を行うサブルーチンであり, 総プロセス数の問い合わせを行う. 第一引数はMPI通信のためのコミュニケータ(注1)"MPI_COMM_WORLD"を指定する. 第二引数(nprocs)に総プロセス数を受け取る.

⑤ MPI_COMM_RANK
MPIのプロセス管理を行うサブルーチンであり, 自プロセスのrank番号の問い合わせを行う. 第二引数(myrank)に自rank番号を受け取る.

⑥ MPI_RECV
MPIの通信処理を行うサブルーチンであり, データの受信をブロッキング(同期)モード(注2)で行う. このサブルーチンは, 一対一通信と呼ばれる分類に属し, 通信相手が必要になる. 送信側が送ったデータを受信するために呼び出される. 第一引数(itmp)は受信データの開始アドレス, 第二引数(1)は受信データの要素数, 第三引数(MPI_INTEGER)は受信データのタイプ, 第四引数(source)は通信相手の rank 番号, 第五引数(tag)はタグ(送受信は同じタグ間で行われる), 第六引数(MPI_COMM_WORLD)はコミュニケータ, 第七引数(status)はメッセージ情報を受け取る.

⑦ MPI_SEND
MPI の通信処理を行うサブルーチンであり, データの送信をブロッキング(同期)モードで行う. このサブルーチンは, 一対一通信と呼ばれる分類に属し, 通信相手が必要になる. 受信側にデータを送信するために呼び出される. 第一引数(isum)は送信データの開始アドレス, 第二引数(1)は送信データの要素数, 第三引数(MPI_INTEGER)は送信データのタイプを示す. 以降の引数はMPI_RECVと同じである.

⑧ MPI_FINALIZE
MPI の終了処理を行うものであり, すべての MPI サブルーチンが呼ばれた最後で呼び出す必要がある.

(注 1)コミュニケータ:通信に参加する MPI プロセスを集めてできたグループにつけられる一種の名前のようなものである. この識別は通信相手のプロセスを特定する場合に必要となる. 利用者は, プログラムにおいて複数のコミュニケータを定義することもできる.

(注 2)ブロッキング(同期)モード:対応する送信手続きと受信手続きのどちらかを先に呼び出した場合に, 両方の呼び出しが揃うまで先行した側の MPI プロセスは待ち合わせを行う処理モードである.

6.3 領域分割法

6.3.1 領域分割法の例

領域分割法とは，図 6.3-1 のようにシミュレーションの対象となる空間を分割し，複数のプロセスで処理を分担して実行するものである．分割した空間の間で，それぞれのプロセスがデータの参照を必要とする場合，MPI を用いてデータの転送を行う．

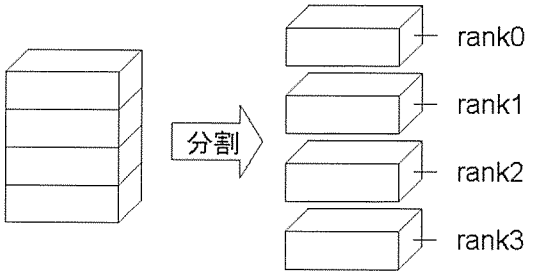

図 6.3-1 領域分割

図 6.3-2 に，差分コードの例を示す．この差分コードの領域を均等に 2 分割し，2 プロセスで実行する場合のコードを図 6.3-3 に示す．

```
do i = 1, 10
  a(i) = i
enddo
do i = 2, 10
  b(i) = a(i) - a(i-1)
enddo
```

図 6.3-2 差分コード

```
do i = 1, 5
  a(i) = i
enddo
do i = 2, 5
  b(i) = a(i) - a(i-1)
enddo
```

rank0

```
do i = 6, 10
  a(i) = i
enddo
do i  = 6, 10
  b(i) = a(i) - a(i-1)
enddo
```

rank1

図 6.3-3 領域分割のコード

図 6.3-3 のように分割する場合，rank 番号 1 は i=6 のときの計算に必要な a(5)のデータが領域外となる．このとき，図 6.3-4 で示されるように a(5)のデータは rank 番号 0 の領域内にあり，rank 番号 1 は rank 番号 0 から a(5)のデータを受け取る必要がある．

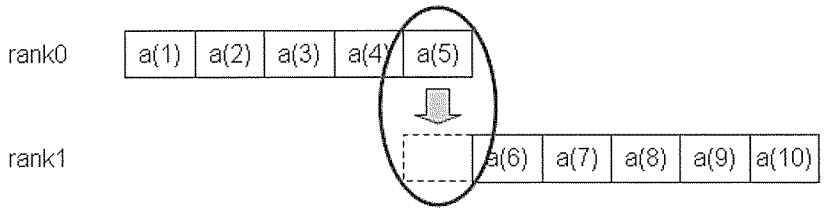

図 6.3-4 境界面で不足するデータのイメージ

6.3.2 領域分割法におけるデータ転送

領域分割法では，自プロセスで演算に必要となるデータが担当する領域の外にある場合，そのデータが含まれる領域を担当するプロセスから，データを受け取る必要がある．このデータ転送がオーバヘッドとなり，並列性能を低下させる要因となる場合がある．

図 6.3-5 はポアソン方程式をヤコビ法で解くコードである．ここでは i, j, k の三次元空間を扱っている．これを用いて領域分割の違いによるデータ転送のコストについて説明する．

```
30: +------>          do loop =1,n
31: |+------>           do k = 2,kmax-1
32: ||+------>            do j = 2,jmax-1
33: |||V--->                do i = 2,imax-1
34: ||||                      u(i,j,k)=s*(uu(i+1,j,k)+uu(i-1,j,k)
35: ||||             &                +uu(i,j+1,k)+uu(i,j-1,k)
36: ||||             &                +uu(i,j,k+1)+uu(i,j,k-1)
37: ||||             &                +a(i,j,k))
38: |||V---                 enddo
39: ||+-----              enddo
40: |+------            enddo
41: |                    err=0.0d0
42: |+------>           do k = 2,kmax-1
43: ||+------>            do j = 2,jmax-1
44: |||V--->                do i = 2,imax-1
45: ||||                      err=err+(uu(i,j,k)-u(i,j,k))*(uu(i,j,k)-u(i,j,k))
46: |||V---                 enddo
47: ||+-----              enddo
48: |+------            enddo
49: |                    if(err .lt. eps) go to 100
50: +-------            end do
51: +------->           do k = 2,kmax-1
52: |+------>            do j = 2,jmax-1
53: ||V----->                do i = 2,imax-1
54: |||                       uu(i,j,k)=u(i,j,k)
55: ||V-----                 enddo
56: |+------              enddo
57: +-------            enddo
58:              100   continue
```

図 6.3-5 ポアソン方程式をヤコビ法で解くコード

三次元空間を j 方向または k 方向で 4 分割することを考える．ただし，配列 uu はサイズ(imax,n,n)の倍精度実数とする．図 6.3-5 の 34-37 行目の配列 uu は 3 次元のそれぞれの方向に差分式となっている．どの方向に分割しても隣接しているデータ(境界面)を参照しているため，MPI を用いてデータ転送する必要

がある. 以下に k 方向, j 方向に分割したときの通信量, 通信回数について説明する.

(1) k 方向に分割したとき

図 6.3-5 の 34-37 行目の配列 uu は分割した空間の境界面のデータを参照(3 次元目が k+1, k-1)しているため, MPI を用いて転送する必要がある. この例では, MPI_SEND, MPI_RECV を用いて転送する. k 方向に分割した場合, 境界は ij 平面となりメモリ上に連続に配置されているので, 1 つの境界面あたり 1 回の通信でデータを送受信することができる. 図 6.3-6 に rank 番号 1 におけるデータ転送を示す. 1 度の通信で(imax×n×8)Byte のデータを転送し, 必要な転送は 2 回のみとなる.

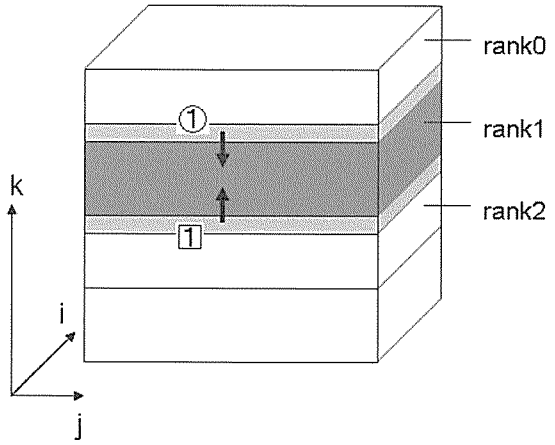

図 6.3-6 k 方向で分割したときのデータ転送

図 6.3-7 は上に示したデータ転送の通信例である(数字は図 6.3-6 と対応). rank 番号 1 は, rank 番号 0, 2 から境界面のデータを受信する必要がある.

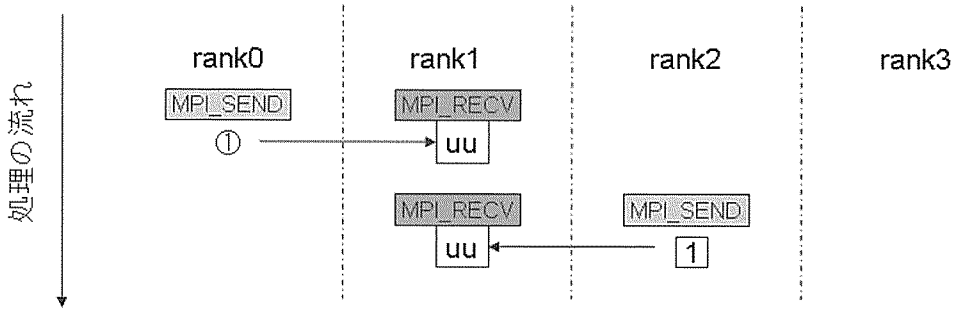

図 6.3-7 k 方向で分割したときの通信

(2) j 方向に分割したとき

配列 uu は分割した空間の境界面のデータを参照(2 次元目が j+1, j-1)しているため, MPI を用いて転送する必要がある. 図 6.3-8 に MPI_SEND, MPI_RECV を使用した場合の rank 番号 1 におけるデータ転送を示す. メモリ上に連続に配置されているデータは i 方向のみであるため, 1 度の通信で転送できるデータは(imax×8)Byte であり, (n×2)回の転送が必要となる.

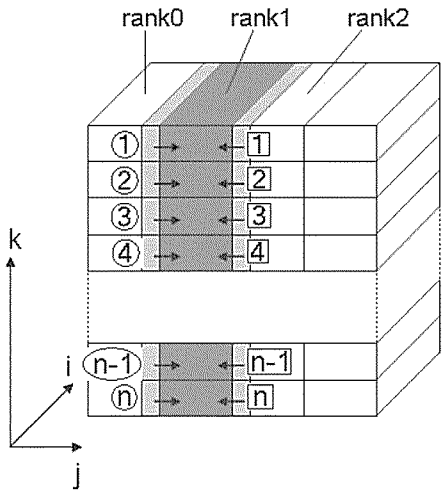

図 6.3-8 j 方向で分割したときのデータ転送

図 6.3-9 は上に示したデータ転送のプロセス間の通信例である(数字は図 6.3-8 と対応). rank 番号 1 は, rank 番号 0, 2 から境界面のデータを受信する必要がある.

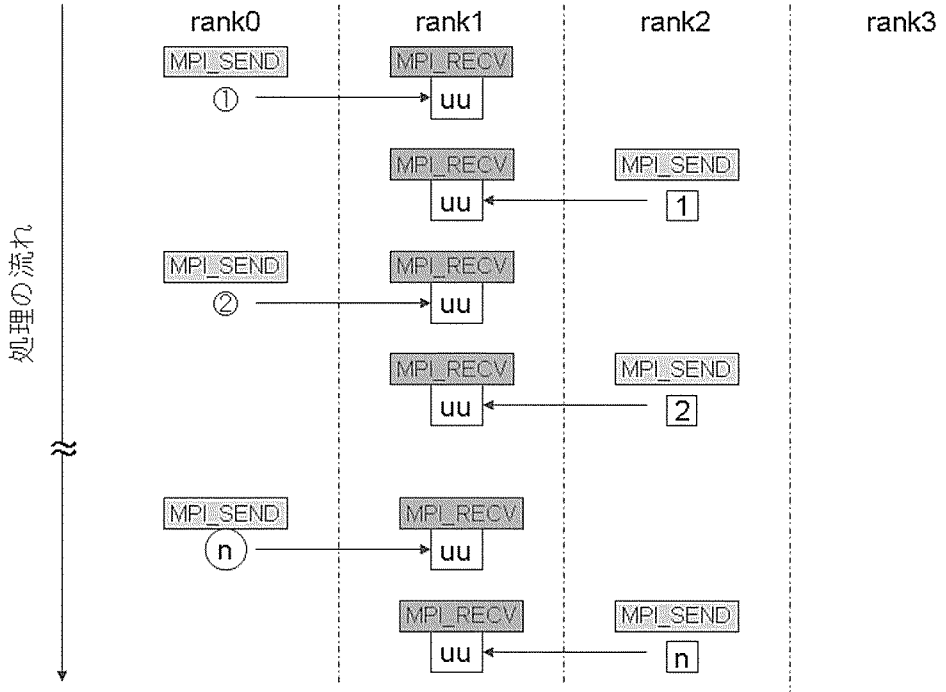

図 6.3-9 j 方向で分割したときの通信例

6.3.3 領域分割法による通信性能

図 6.3-5 にあるポアソン方程式をヤコビ法で解くコードを, 問題サイズ「imax=1280, jmax=128, kmax=128」として性能評価を行う. このコードを MPI で分散並列化し, 16 プロセスで並列実行するときの通信量, 呼び出し回数, 総通信量を表 6.3-1 に示す. jmax=kmax であるため, j, k の分割方向によらず総通信量は同じである.

表 6.3-1 通信量，呼び出し回数，総通信量

	1 回あたりの MPI 通信量	呼び出し回数	総通信量
k 方向に分割	約 1.25MB	20,000	約 25,000MB
i 方向に分割	約 10KB	2,560,000	約 25,000MB

上記条件で，SX-9 とExpress5800 それぞれで実行する．図 6.3-10 に分割方向の違いによる通信時間のグラフを示す．総通信量は同じであるが，SX-9，Express5800 ともに，通信回数の少ない k 方向の分割の方が通信時間は短いことがわかる．このように並列性能の向上のためには，通信回数が少なく，また 1 度の通信で多くのデータを送受信できる適切な分割方法を検討することが重要になる．

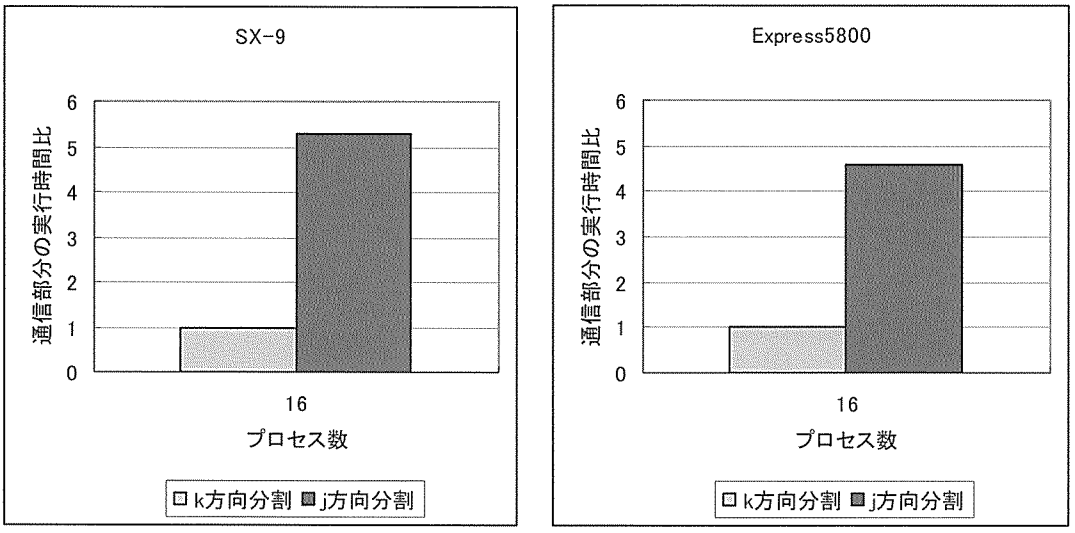

図 6.3-10 分割方向の違いによる通信時間

6.4 領域分割法における MPI コード例

本節では，6.3 節のポアソン方程式を領域分割法によって MPI 化した実際のコードについて説明する．

6.4.1 MPI による隣接領域間の転送

6.3 節でも述べたが，図 6.4-1 の 34-37 行目の配列 uu は分割した空間の境界面のデータを参照している差分式であるため，隣接領域間の転送が必要である．

```
30: +------>          do loop =1,n
31: |+------>           do k = 2,kmax-1
32: ||+----->            do j = 2,jmax-1
33: |||V---->              do i = 2,imax-1
34: ||||                     u(i,j,k)=s*(uu(i+1,j,k)+uu(i-1,j,k)
35: ||||              &                 +uu(i,j+1,k)+uu(i,j-1,k)
36: ||||              &                 +uu(i,j,k+1)+uu(i,j,k-1)
37: ||||              &                 +a(i,j,k))
38: |||V---              enddo
39: ||+-----           enddo
40: |+------          enddo
 :
57: +------          enddo
```

図 6.4-1 ポアソン方程式における差分式（逐次処理コード）

図 6.4-2 に領域を分割したとき，差分式の演算で必要な境界面のデータ通信を示す．k 方向に領域を 4 分割すると，①～④の切り出した領域を各プロセスが分担する．分割した領域には隣接した空間の境界面のデータをそれぞれ冗長に持つ必要がある．具体的には，①は B，②は A と D，③は C と F，④は E の境界面のデータを保有する．ただし，これら境界面は隣接したプロセスが演算結果を更新するため，自プロセスでは更新前の古いデータを保持することなる．そのため，各プロセスは隣り合う rank 番号に対して更新されたデータの送受信を行う必要がある．

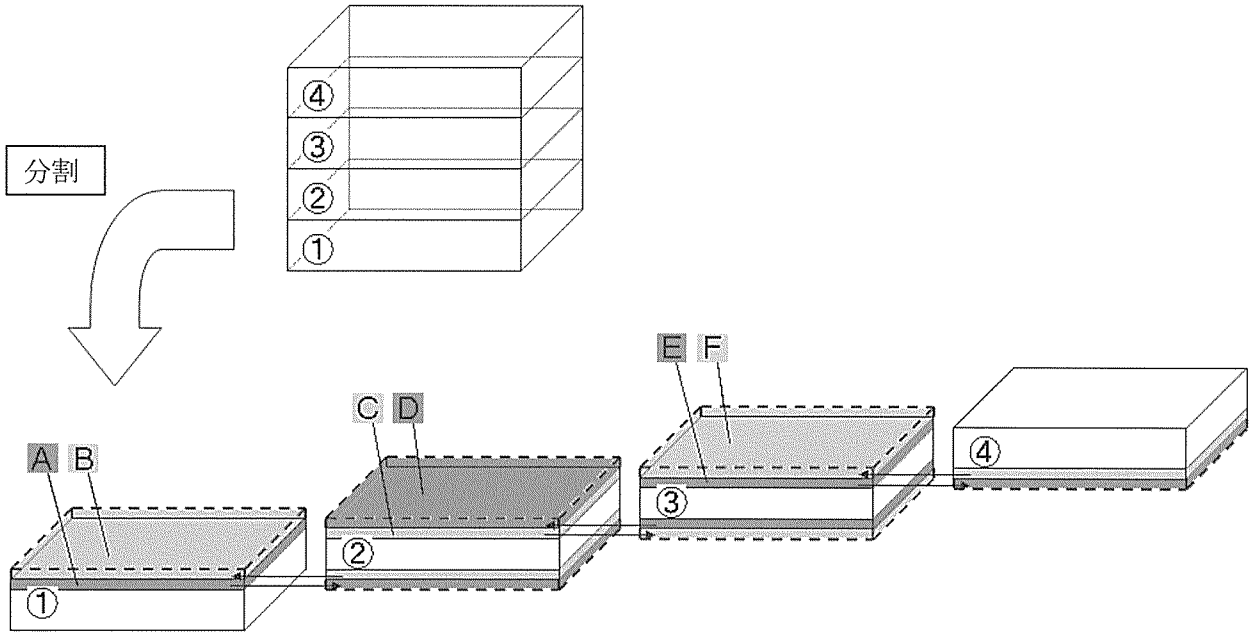

図 6.4-2 領域分割とデータ通信

図 6.4-3 に MPI_SEND，MPI_RECV を用いたコード例を示す．この例では分割する 3 次元目のサイズは実行するプロセス数で割り切れるものとする．21-59 行目では，k 方向に分割したときの各プロセスが必要とするデータ(作業配列，ループ長とループの始点，終点)を作成し，71-80 行目では，分割された領域内でポアソン方程式を解く．94-117 行目で更新された境界面のデータを転送する．

```
 :  !各プロセスが担当する3次元目の大きさを計算
 21:                    kmax = kmax/nprocs
 :  !各プロセスにおける配列uの3次元目の始点を計算
 23:                      ukst =myrank*kmax
 :  !境界面のデータを格納する領域を持つ配列uu,各プロセスの配列uの値を保持する作業配列wuの確保
 29:                    allocate(uu(imax, jmax, 0:kmax+1), wu(imax, jmax, kmax))
 :  !各プロセス毎の始点, 終点の定義
 50:                    if(myrank .eq. 0)then
 51:                      kst=2
 52:                    else
 53:                      kst=1
 54:                    endif
 55:                    if(myrank .eq. nprocs-1)then
 56:                      ked=kmax-1
 57:                    else
 58:                      ked=kmax
 59:                    endif
 :
 71: +------->          do loop =1,n
 72: |+------>            do k = kst,ked
 73: ||+----->              do j = 2,jmax-1
 74: |||V---->                do i = 2,imax-1
 75: ||||                       wu(i,j,k)=s*(uu(i+1,j,k)+uu(i-1,j,k)
 76: ||||              &                    +uu(i,j+1,k)+uu(i,j-1,k)
 77: ||||              &                    +uu(i,j,k+1)+uu(i,j,k-1)+a(i,j,k))
 78: |||V---                  enddo
 79: ||+-----               enddo
 80: |+------              enddo
 :  !隣接領域間の転送
 94: |                    if(myrank.ne.0)then
 95: |                      dest=myrank-1
 96: |                      tag=myrank
 97: |                      call MPI_SEND(wu(1,1,kst), imax*jmax,MPI_DOUBLE_PRECISION,
 98: |                   &                dest,tag,MPI_COMM_WORLD,IERR)
 99: |                    endif
100: |                    if(myrank.lt.nprocs-1)then
101: |                      source=myrank+1
102: |                      tag=myrank+1
103: |                      call MPI_RECV(uu(1,1,ked+1), imax*jmax,MPI_DOUBLE_PRECISION,
104: |                   &                source,tag,MPI_COMM_WORLD,status,IERR)
105: |                    endif
106: |                    if(myrank.ne.0)then
107: |                      source=myrank-1
108: |                      tag=myrank-1
109: |                      call MPI_RECV(uu(1,1,kst-1), imax*jmax,MPI_DOUBLE_PRECISION,
110: |                   &                source,tag,MPI_COMM_WORLD,status,IERR)
111: |                    endif
112: |                    if(myrank.lt.nprocs-1)then
113: |                      dest=myrank+1
114: |                      tag=myrank
115: |                      call MPI_SEND(wu(1,1,ked), imax*jmax,MPI_DOUBLE_PRECISION,
116: |                   &                dest,tag,MPI_COMM_WORLD,IERR)
117: |                    endif

130: +-------          enddo
```

図 6.4-3 ポアソン方程式における差分式(MPIコード)

6.4.2 収束判定のための総和演算

図 6.4-4 は，ヤコビ法の収束判定を行っている箇所を抜き出したものである．49 行目の変数 err の値によって収束判定を行っている．

```
30: +------->          do loop =1,n
 :
41: |                   err=0.0d0
42: |+------>           do k = 2,kmax-1
43: ||+----->             do j = 2,jmax-1
44: |||V---->               do i = 2,imax-1
45: ||||                      err=err+(uu(i,j,k)-u(i,j,k))*(uu(i,j,k)-u(i,j,k))
46: |||V---                 enddo
47: ||+-----              enddo
48: |+------              enddo
49: |                     if(err .lt. eps) go to 100
 :
57: +-------              enddo
58:              100  continue
```

図 6.4-4 ヤコビ法の収束判定（逐次処理コード）

MPI コードでは図 6.4-4 の 42-48 行目のループは分割されているため，変数 err は部分和となる．各プロセスの部分和を集計し，総和を求める必要がある．6.2 節の総和計算では，一対一通信の MPI_RECV と MPI_SEND のサブルーチンを使用して集計したが，ここでは総和演算を MPI の集団通信のサブルーチンで集計する例を図 6.4-5 に示す．

```
 71: +------->          do loop =1,n
 :
 81: |                   err=0.0d0
 82: |+------>          do k = kst,ked
 83: ||+----->             do j = 2,jmax-1
 84: |||V---->               do i = 2,imax-1
 85: ||||                      err=err+(uu(i,j,k)-wu(i,j,k))
 86: ||||              &              *(uu(i,j,k)-wu(i,j,k))
 87: |||V---                 enddo
 88: ||+-----              enddo
 89: |+------              enddo
 90: |                     CALL MPI_ALLREDUCE(err,all_err,1,MPI_DOUBLE_PRECISION,     !①
 91: |                   &                    MPI_SUM,MPI_COMM_WORLD,IERR)
 92: |                     if (myrank.eq.0) write(*,*) all_err,loop
 93: |                   if(all_err .lt. eps) go to 100
 :
130: +-------           enddo
131:
132:               100 continue
```

図 6.4-5 ヤコビ法の収束判定（MPI コード）

この例で使用した MPI サブルーチンについて以下に示す．

① MPI_ALLREDUCE

MPI の集団通信を行うサブルーチンであり，複数のプロセスからそれぞれのデータを受信し，リダクション演算を行う．集団通信は，同一コミュニケータの全プロセスが対象になる．第五引数に MPI_SUM を与えると，第一引数で与えられた各プロセスのデータの総和を第二引数で受け取る．

6.5 デッドロック

MPI プログラムの実行時には，プログラムの書き方によって各プロセスが送受信待ち状態となり次の処理へ進めなくなる場合がある．この状態をデッドロックと言う．図 6.5-1 はデッドロックが発生する場合のコードである．

```
do i = 0,nprocs-1
  if(myrank==i)then
    k=mod(i+1,nprocs)
    itag=k
    call MPI_SEND(b(1+i*it),it,MPI_INTEGER,k,itag,MPI_COMM_WORLD,ierr)
  endif
enddo

do i = 0,nprocs-1
  if(myrank==i)then
    k=mod(i-1+nprocs,nprocs)
    itag=i
    call MPI_RECV(b(1+i*it),it,MPI_INTEGER,k,itag,MPI_COMM_WORLD,status,ierr)
    endif
enddo
```

図 6.5-1 デッドロックのコード例

MPI_SEND は 6.2 節で述べたようにブロッキングモードで通信を行うため，データの受信が完了しないと次の処理に進めない．つまり，MPI_SEND に対して，受信側プロセスで MPI_RECV が実行されるまで処理は中断される．図 6.5-2 は図 6.5-1 を 2 並列で実行したときのイメージである．この例では，rank 番号 0 は rank 番号 1 が MPI_RECV を呼ぶまで MPI_SEND で待ち続け，rank 番号 1 は rank 番号 0 が MPI_RECV を呼ぶまで MPI_SEND で待ち続けることになり処理が進まなくなる．

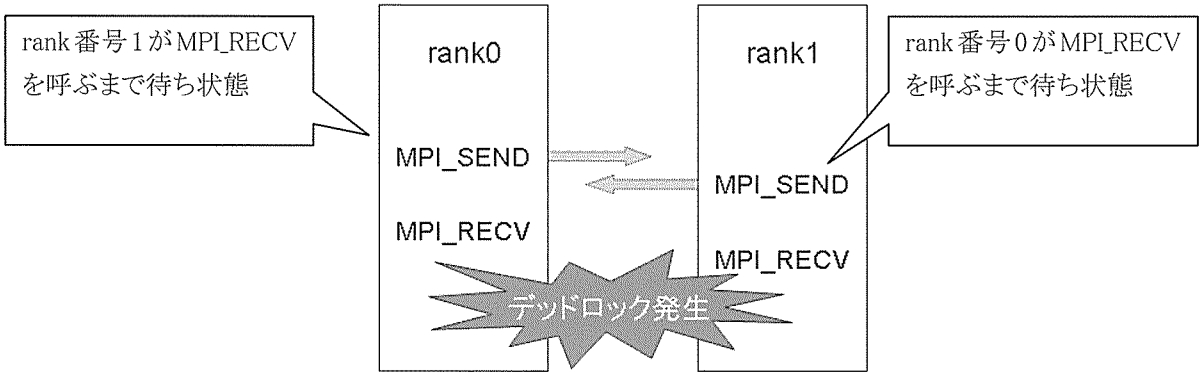

図 6.5-2 デッドロック発生のイメージ

```
do i = 0,nprocs-1
  if(myrank==i)then
    k=mod(i+1,nprocs)
    itag=k
    call MPI_ISEND(b(1+i*it),it,MPI_INTEGER,k,itag,MPI_COMM_WORLD,request,ierr)      !①
  endif
enddo

do i = 0,nprocs-1
  if(myrank==i)then
    k=mod(i-1+nprocs,nprocs)
    itag=i
    call MPI_RECV(b(1+i*it),it,MPI_INTEGER,k,itag,MPI_COMM_WORLD,status,ierr)
  endif
enddo

call MPI_WAIT(request,status,ierr)                                                  !②
```

図 6.5-3 正常動作(デッドロックが発生しない)例

図 6.5-3 にデッドロックを発生させずに実行するコードを示す. 図 6.5-1 では MPI_SEND を使用していたのに対し, 図 6.5-3 では MPI_ISEND を使用している. MPI_ISEND はノンブロッキングモード(注)で通信を行うため, 送信バッファが再利用可能になるのを待たずに次の処理へ進むことができる. ただし, MPI_WAIT を指定し, 通信が完了したことを待ち合わせる必要がある. 図 6.5-4 は図 6.5-3 を 2 並列で実行したときのイメージである.

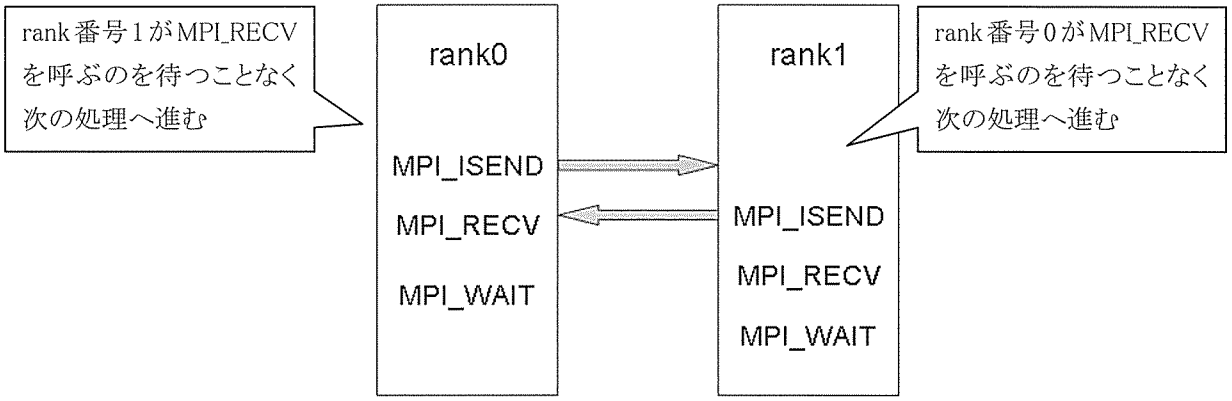

図 6.5-4 正常動作のイメージ

図 6.5-3 で使用した MPI サブルーチンについて以下に示す.

① MPI_ISEND

MPI の通信処理を行うサブルーチンであり, データの送信をノンブロッキング(非同期)モードで行う. このサブルーチンは一対一通信と呼ばれる分類に属し, 通信相手が必要になる. 受信側にデータを送信するために呼び出される. MPI_WAIT とペアで使用する. 第一引数から第六引数は, MPI_SEND と同じである. 第七引数(request)は通信識別子を示す.

② MPI_WAIT

ノンブロッキング通信の完了の待ち合わせを行う. 一対一ノンブロッキング通信を使用した場合にペアで使用する. 第一引数(request)に通信識別子, 第二引数(status)にメッセージ情報を示す.

(注)ノンブロッキング(非同期)モード:送信または受信の操作が完了する前に手続きから戻る. ノンブロッキング通信を使用する場合は, 送受信操作が完了する前に送信または受信領域の内容を変更してはならない.

6.6 SX-9 の最適化事例

6.6.1 通信方法の見直し

適切な通信サブルーチンを選択することにより, 通信時間を短縮することができる. 本節では, 集団通信である MPI_SCATTERV と MPI_GATHERV を単方向通信である MPI_GET と MPI_PUT に置き換える例を示す. MPI_GET, MPI_PUT は, MPI プロセスが単独で他の MPI プロセスに対してデータの送受信を行うことができる. このため, すべてのプロセスが同時に通信を行う集団通信 MPI_SCATTERV, MPI_GATHERV に比べて実行時間を短縮することができる.

図 6.6-1 は rank 番号 0 のプロセスが持つデータを他の rank 番号のプロセスへ送り, 処理を行った後にそれらのデータを rank 番号 0 のプロセスへ送り返すコードである. 図 6.6-1 の通信のイメージを図 6.6-2 に示す.

```
   if(myrank.eq.0) then
     call MPI_SCATTERV(data,iscnt,idisp,MPI_REAL8,                    !①
    &                  MPI_IN_PLACE,idummy,idummy,0,
    &                  MPI_COMM_WORLD,ierr)
    else
     im=nz/nprocs*myrank+1
     call MPI_SCATTERV(data,iscnt,idisp,MPI_REAL8,                    !①
    &                  data(1,1,im),isize,MPI_REAL8,0,
    &                  MPI_COMM_WORLD,ierr)
    endif
     :
    if(myrank.eq.0) then
     call MPI_GATHERV(MPI_IN_PLACE,idummy,idummy,                     !②
    &                  data,ircnt,idisp,MPI_REAL8,0,
    &                  MPI_COMM_WORLD,ierr)
    else
     call MPI_GATHERV(data(1,1,im),iscnt,MPI_REAL8,                   !②
    &                  data,ircnt,idisp,MPI_REAL8,0,
    &                  MPI_COMM_WORLD,ierr)
    Endif
```

図 6.6-1 MPI_SCATTERV と MPI_GATHERV を用いた通信コード

この例で使用した MPI サブルーチンについて以下に示す.

① MPI_SCATTERV

MPI の集団通信を行うサブルーチンであり, 同一コミュニケータ内の 1 つのプロセスがデータを送信し, 全プロセスが受信を行う. 第一引数は送信データの先頭アドレス, 第二引数は送信するデータ要素数(プロセス毎), 第三引数は送信データの先頭後レスからの変位(プロセス毎), 第四引数(MPI_REAL8)は送信するデータの型, 第四引数は受信データの先頭アドレス, 第五引数(MPI_REAL8)は受信するデータの要素数, 第六引数は受信するデータの型, 第七引数(0)は送信元プロセスの rank 番号を示す.

② MPI_GATHERV

MPI の集団通信を行うサブルーチンのであり, 同一コミュニケータ内のすべてのプロセスがデータを送信し, 1 つのプロセスがそれらのデータの受信を行う. 第一引数は送信データの先頭アドレス, 第二引

数は送信するデータの要素数，第三引数(MPI_REAL8)は送信するデータの型，第四引数は受信データの先頭アドレス，第五引数は受信するデータの要素数(プロセス毎)，第六引数は受信データの先頭アドレスからの変位(プロセス毎)，第七引数(MPI_REAL8)は受信するデータの型，第七引数(0)は送信先プロセスの rank 番号を示す．

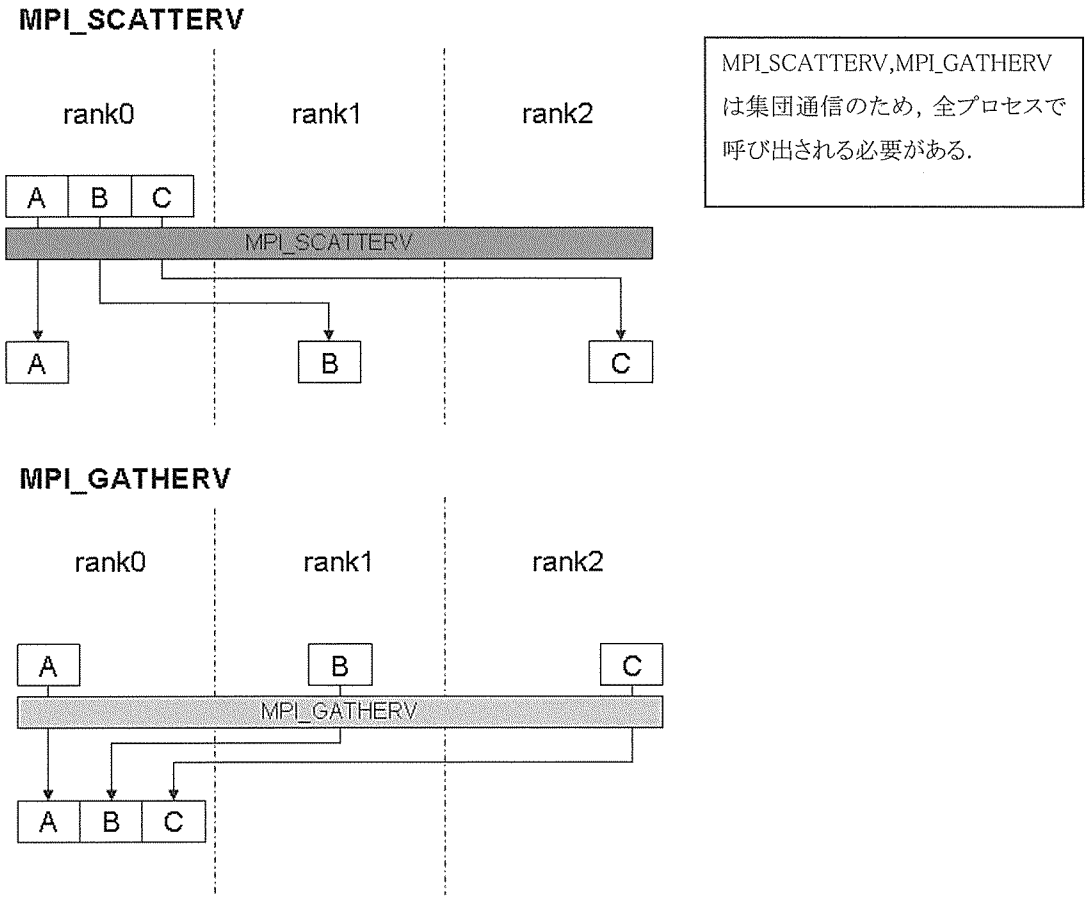

図 6.6-2　MPI_SCATTERV と MPI_GATHERV を用いた通信

次に MPI_SCATTERV，MPI_GATHERV を MPI_GET，MPI_PUT に置き換える．MPI_GET，MPI_PUT を用いた通信イメージを図 6.6-3 に，コードを図 6.6-4 に示す．

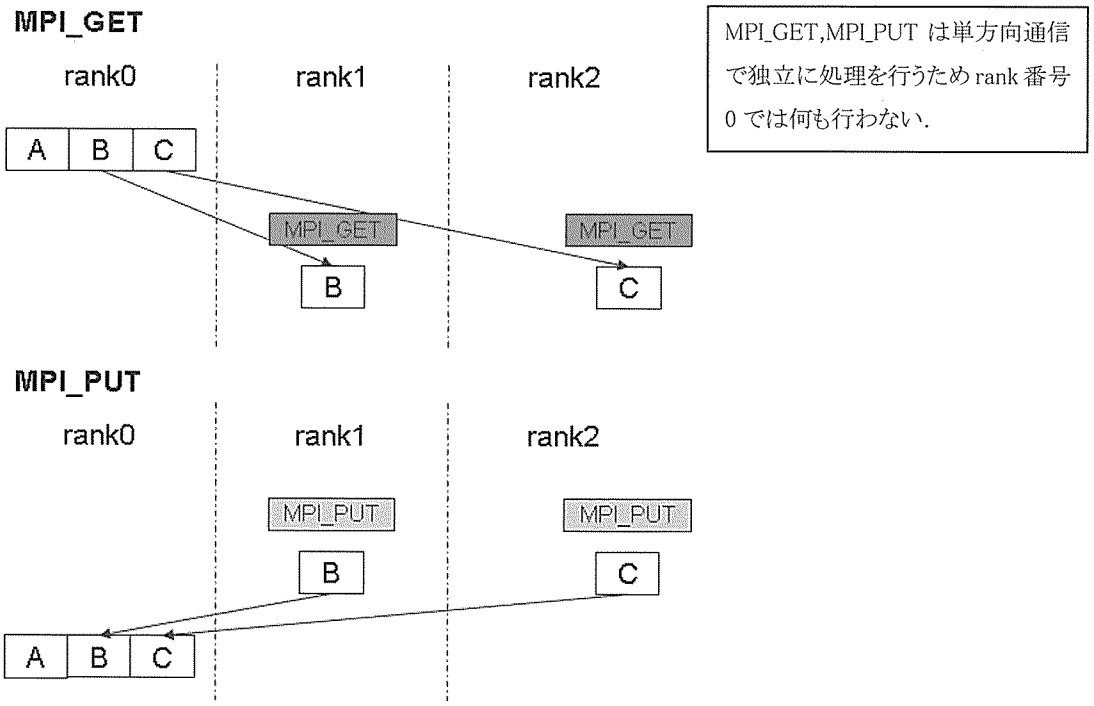

図 6.6-3 MPI_GET と MPI_PUT を用いた通信

```
  integer(kind=MPI_ADDRESS_KIND) idisp                              !①

  isize=8*nx*ny*nz
  call MPI_WIN_CREATE(data, isize, 8, MPI_INFO_NULL,                !②
 &                    MPI_COMM_WORLD, win, ierr)
  call MPI_WIN_FENCE(0, win, ierr)                                  !③

  idisp=nx*ny*nz/nprocs*myrank
  call MPI_WIN_FENCE(0, win, ierr)                                  !③
  if(myrank.ne.0) then
    call MPI_GET(data(1,1,nz/nprocs*myrank+1), nx*ny*nz/nprocs,     !④
 &               MPI_REAL8, 0, idisp, nx*ny*nz/nprocs,
 &               MPI_REAL8, win, ierr)
  endif
  call MPI_WIN_FENCE(0, win, ierr)                                  !③
   :
  call MPI_WIN_FENCE(0, win, ierr)                                  !③
  if(myrank.ne.0) then
    call MPI_PUT(data(1,1,nz/nprocs*myrank+1), nx*ny*nz/nprocs,     !⑤
 &               MPI_REAL8, 0, idisp, nx*ny*nz/nprocs,
 &               MPI_REAL8, win, ierr)
  endif
  call MPI_WIN_FENCE(0, win, ierr)                                  !③

  call MPI_WIN_FREE(win, ierr)                                      !⑥
```

図 6.6-4 MPI_PUT と MPI_GET を用いた通信コード

この例で使用した MPI の宣言部分とサブルーチンについて以下に示す.

① integer(kind＝MPI_ADDRESS_KIND)

下記, MPI_GET, MPI_PUT の第五引数で使用する変数は, この型を宣言する.

② MPI_WIN_CREATE

MPI の単方向通信は，許可したメモリ領域以外，他プロセスからのアクセスを禁止している．この許可したメモリ領域のことをウィンドウと言い，MPI_WIN_CREATE はこのウィンドウの登録を行うサブルーチンである．第一引数はウィンドウに登録するデータの先頭アドレス，第二引数はウィンドウのサイズ（単位は byte），第三引数はウィンドウ内の 1 つのデータのサイズ（単位は byte），第四引数（MPI_INFO_NULL）は info 引数，第六引数は登録されるウィンドウを示す．

③ MPI_WIN_FENCE

単方向通信 MPI の通信処理の同期をとるためのサブルーチンである．ウィンドウをアクセスする可能性のあるすべてのプロセスで呼び出される必要がある．

④ MPI_GET

MPI の単方向通信を行うサブルーチンであり，転送対象プロセスのウィンドウ領域からデータの読込みを行う．第一引数は受信先の先頭アドレス，第二引数は受信するデータ数，第三引数は受信するデータの型，第四引数は送信元の rank 番号，第五引数は登録したウィンドウの先頭からの変位，第六引数は送信するデータ数，第七引数は送信するデータの型を示す．

⑤ MPI_PUT

MPI の単方向通信を行うサブルーチンであり，転送対象プロセスのウィンドウ領域に対してデータの書込みを行う．第一引数は送信するデータの先頭アドレス，第二引数は送信するデータ数，第三引数は送信するデータの型，第四引数は受信先の rank 番号，第五引数は登録したウィンドウの先頭からの変位，第六引数は受信するデータ数，第七引数は送信するデータの型を示す．

⑥ MPI_WIN_FREE

ウィンドウの登録を抹消するためのサブルーチンである．第一引数は登録を抹消するウィンドウを示す．

SX-9 におけるそれぞれの通信時間は図 6.6-5 のとおりである．なお通信に使用する配列 data のサイズは約 8GB，プロセス数は 4 プロセスである．MPI の集団通信から単方向通信へ変更することで 1.27 倍の性能向上が得られる．

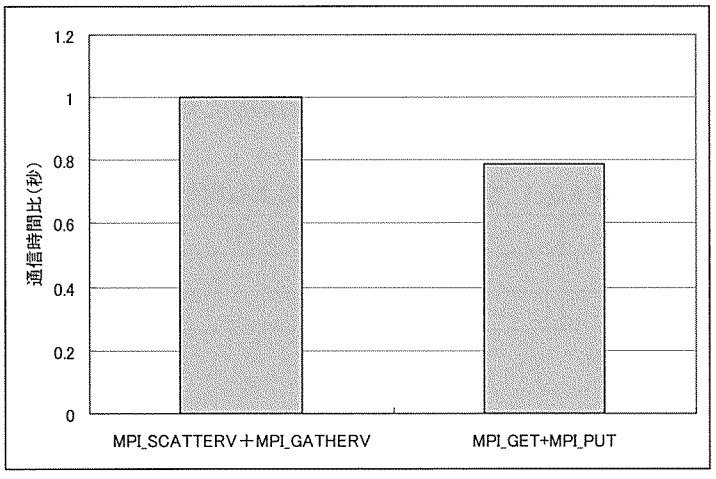

図 6.6-5 通信の違いによる性能比

6.6.2 SX-9 のグローバルメモリ機能

SX-9 では通常のユーザプログラムが使用するメモリ空間はローカルメモリにある. MPI がデータ通信をする際には, 図 6.6-6 に示すように, ローカルメモリ上にある転送データをグローバルメモリの通信バッファへコピーし, MPI システム通信バッファ間で転送が行われる. 受信側で MPI システム通信バッファからローカルメモリのユーザ領域へコピーが完了した時点でデータ通信が終了する. SX-9 では, グローバルメモリ機能を提供しており, 図 6.6-7 に示すようにユーザプログラムが使用するメモリ空間をグローバルメモリ上へ割り付けておくことができる. このことにより, ローカルメモリとグローバルメモリ間のメモリコピーを行うことなく, グローバルメモリ空間にあるユーザ領域のデータを直接転送することができる.

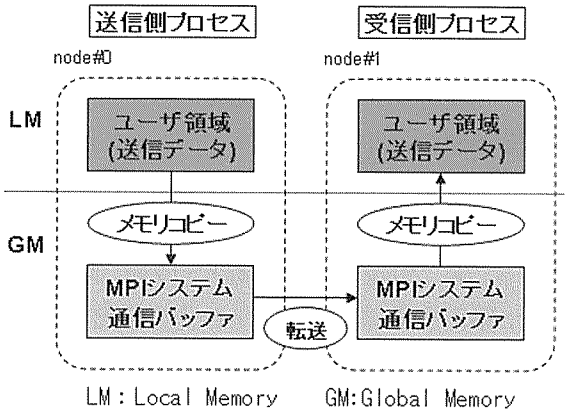

図 6.6-6 通常の MPI データ転送手順

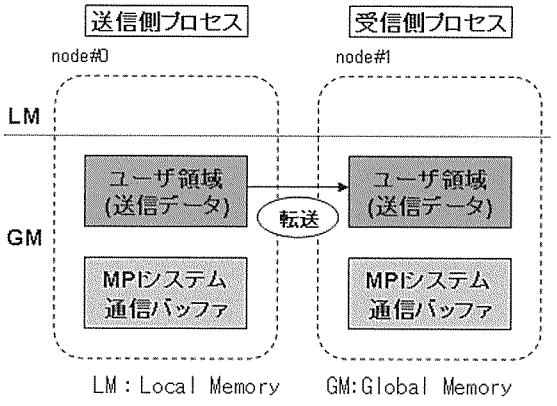

図 6.6-7 グローバルメモリ領域を使用したMPI データ転送手順

グローバルメモリ機能を使用するためには, 通信するデータを格納する配列を動的(allocatable 属性)に確保し, コンパイルオプション「-gmalloc」を指定する. ただし, 配列を通信するごとに動的に確保するような場合, 配列の割当/解放の処理がオーバヘッドとなりグローバルメモリを用いた効果を打ち消す場合がある. このためグローバルメモリの動的な割当/解放は最小限に止めることが必要である. 表 6.6-1 に, SX-9 においてグローバルメモリ機能を使用した場合としない場合のデータ転送時間を示す. これは, 約 100MB のデータを MPI_ALLGATHER でデータ転送した例である. グローバルメモリ機能を使用することで, 2 ノード(32 プロセス)使用時に 2.77 倍, 4 ノード(64 プロセス)使用時に 2.67 倍の性能向上が得られることがわかる.

表 6.6-1 グローバルメモリによる実行時間の比較

	2 ノード(32 プロセス)	4 ノード(64 プロセス)
グローバルメモリ機能なし	162.031sec	327.985sec
グローバルメモリ機能あり	58.441sec	122.542sec

6.7 SX-9 における MPI 性能情報採取方法

本節では SX-9 における MPI 性能情報の採取方法について説明する.

(1) MPIPROGINF

MPI/SX では各 MPI プロセスの性能情報を採取する機能がある. この機能によって MPI プロセス毎, または全 MPI プロセスの情報を集計編集して表示させることができる. 性能情報の表示は, MPI_COMM_WORLD(MPI_UNIVERSE=0)の rank 番号 0 における MPI プロセスの標準エラー出力に対して行われる. 使用するにはプログラム実行時に環境変数 MPIPROGINF を指定する. MPIPROGINF の値とその動作は以下, 表 6.7-1 のとおりである.

表 6.7-1 MPIPROGINF の値

NO	性能情報を出力しない(既定値).
YES	基本情報を集約形式で出力する.
DETAIL	詳細情報を集約形式で出力する.
ALL	基本情報を拡張形式で出力する.
ALL_DETAIL	詳細情報を拡張形式で出力する.

```
MPI Program Information:
========================
Note: It is measured from MPI_Init till MPI_Finalize.
      [U,R] specifies the Universe and the Process Rank in the Universe.

                                        (a)                 (b)             (c)
Global Data of 16 processes :       Min [U,R]           Max [U,R]       Average
============================

Real   Time (sec)           :     1.350 [0,15]        1.577 [0,0]       1.464
User   Time (sec)           :     1.267 [0,13]        1.324 [0,8]       1.308
System Time (sec)           :     0.008 [0,2]         0.012 [0,13]      0.008
Vector Time (sec)           :     0.735 [0,0]         0.766 [0,8]       0.752
Instruction Count           :  192390742 [0,0]      200269828 [0,1]   197702659
Vector Instruction Count    :    4788036 [0,0]        5041807 [0,1]     4979275
Vector Element Count        :   75304093 [0,8]       78786404 [0,1]    77034023
FLOP Count                  :     200402 [0,2]         200411 [0,0]      200403
MOPS                        :    199.609 [0,0]        210.707 [0,15]    206.323
MFLOPS                      :      0.151 [0,8]          0.158 [0,13]      0.153
Average Vector Length       :     14.988 [0,8]         15.854 [0,15]     15.472
Vector Operation Ratio (%)  :     28.193 [0,8]         28.841 [0,15]     28.555
Memory size used (MB)       :   1095.745 [0,0]       1095.745 [0,0]    1095.745
Global Memory size used (MB):     64.000 [0,0]         64.000 [0,0]      64.000
MIPS                        :    146.014 [0,0]        153.770 [0,15]    151.210
Instruction Cache miss (sec):      0.080 [0,3]          0.093 [0,0]       0.085
Operand     Cache miss (sec):      0.012 [0,8]          0.030 [0,0]       0.013
Bank Conflict Time
  CPU Port Conf.       (sec):      0.001 [0,7]          0.001 [0,13]      0.001
  Memory Net. Conf.    (sec):      0.578 [0,0]          0.608 [0,8]       0.595
```

図 6.7-1 MPIPROGINF の出力例(DETAIL 指定時)

表示される項目の意味は以下のとおりである. その他の項目はプログラム実行解析情報(PROGINF)と同じとなる.

(a):すべての MPI プロセスを対象に, 基本情報または詳細情報について集計した最小値と rank 番号

(b):すべてのMPIプロセスを対象に，基本情報または詳細情報について集計した最大値とrank番号

(c):すべてのMPIプロセスを対象に，基本情報または詳細情報について集計した平均値

MPIPROGINF の各項目で最大値と最小値の差が大きい場合，処理のインバランス(注)が発生している可能性がある．領域が均等に分割されていてもプロセスごとの処理量が異なるとインバランスが発生する．領域の分割方法の変更などの検討が必要である．

(注)インバランス:各プロセスに割り当てる処理が不均衡な状態を言う．インバランスの場合，プロセス間で同期を取るタイミングで，処理の長いプロセスを待ち合わせする必要があり，先に処理を終えたプロセスに待ち時間が発生する．

(2) MPICOMMINF

MPI/SX では各 MPI プロセスの MPI 通信情報を採取する機能がある．この機能によって全 MPI 手続き実行所要時間，MPI 通信待ち合わせ時間，送受信データ総量，および主要 MPI 手続き呼び出し回数を表示することができる．表示は，MPI_COMM_WORLD (MPI_UNIVERSE=0) の rank 番号 0 における MPI プロセスの標準エラー出力に対して行われる．使用するにはプログラム実行時に環境変数 MPICOMMINF を指定する．MPICOMMINF の値とその動作は以下，表 6.7-2 のとおりである．

表 6.7-2 MPICOMMINF の値

NO	通信情報を出力しない(既定値).
YES	最小値，最大値，および平均値を表示する.
ALL	最小値，最大値，平均値，および各プロセス毎の値を表示する.

```
MPI Communication Information:
------------------------------

Real MPI Idle Time (sec)    :      0.092 [0,8]        0.098 [0,1]        0.095   (a)
User MPI Idle Time (sec)    :      0.089 [0,13]       0.098 [0,1]        0.095   (b)
Total real MPI Time (sec)   :      1.302 [0,15]       1.532 [0,0]        1.419   (c)
Send        count           :          0 [0,0]            0 [0,0]            0
Recv        count           :          0 [0,0]            0 [0,0]            0
Barrier     count           :          0 [0,0]            0 [0,0]            0
Bcast       count           :          0 [0,0]            0 [0,0]            0
Reduce      count           :          0 [0,0]            0 [0,0]            0
Allreduce   count           :          0 [0,0]            0 [0,0]            0
Scan        count           :          0 [0,0]            0 [0,0]            0
Exscan      count           :          0 [0,0]            0 [0,0]            0
Redscat     count           :          0 [0,0]            0 [0,0]            0
Gather      count           :          0 [0,0]            0 [0,0]            0
Gatherv     count           :          0 [0,0]            0 [0,0]            0
Allgather   count           :     100000 [0,0]       100000 [0,0]       100000
Allgatherv  count           :          0 [0,0]            0 [0,0]            0   (d)
Scatter     count           :          0 [0,0]            0 [0,0]            0
Scatterv    count           :          0 [0,0]            0 [0,0]            0
Alltoall    count           :          0 [0,0]            0 [0,0]            0
Alltoallv   count           :          0 [0,0]            0 [0,0]            0
Alltoallw   count           :          0 [0,0]            0 [0,0]            0
Number of bytes sent        :          0 [0,0]            0 [0,0]            0
Number of bytes recv        :          0 [0,0]            0 [0,0]            0
Put         count           :          0 [0,0]            0 [0,0]            0
Get         count           :          0 [0,0]            0 [0,0]            0
Accumulate  count           :          0 [0,0]            0 [0,0]            0
Number of bytes put         :          0 [0,0]            0 [0,0]            0
Number of bytes got         :          0 [0,0]            0 [0,0]            0
Number of bytes accum       :          0 [0,0]            0 [0,0]            0
```

図 6.7-2 MPICOMMINF の出力例(YES 指定時)

表示される各項目の意味は以下のとおりである.

(a):プログラム実行に要した実 MPI 通信待ち合わせ時間
(b):プログラム実行に要したCPU時間の内, ユーザルーチン実行に要したMPI通信待ち合わせ時間
(c):全 MPI サブルーチン実行時間
(d):主要 MPI サブルーチン呼び出し回数

ただし, 本機能を利用する場合には, プロファイル版MPIライブラリをリンクする必要がある. プロファイル版 MPI ライブラリは, MPI プログラムのコンパイル/リンク用シェルスクリプト(mpif90)の-mpitrace, -mpiprof, -ftrace いずれかのオプション指定によりリンクされる.

主要 MPI サブルーチン呼び出し回数と次節 6.7(3)の FTRACE とを合わせて, 通信性能を分析することができる. 詳しくは 6.7(3)を参照のこと.

(3) FTRACE

4.3.1(4)にて, 逐次処理プログラムにおける FTRACE について説明した. MPI プログラムで本機能を使用すると, 逐次処理プログラムと同様の情報に加え, 手続きごと, あるいはユーザ指定リージョンの MPI 通信性能情報を採取することができる. 使用するには, 測定対象のソースプログラムをオプション -ftrace を指定してコンパイルする. 環境変数 F_FTRACE の値とその動作は以下, 表 6.7-3 のとおりで

ある.

表 6.7-3 F_FTRACE の値

YES	プログラム終了直後に解析リストを標準エラー出力ファイルに形式2で出力することを指定する(FMT2 と同じ).
FMT0	プログラム終了直後に解析リストを標準エラー出力ファイルに形式0(注1)で出力することを指定する.
FMT1	プログラム終了直後に解析リストを標準エラー出力ファイルに形式1(注2)で出力することを指定する.
FMT2	プログラム終了直後に解析リストを標準エラー出力ファイルに形式2(注3)で出力することを指定する.
NO	プログラム終了直後に解析リストを標準エラー出力ファイルに出力しないことを指定する(既定値).

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

test             1  1.320(100.0)  1320.361   199.8     0.2 28.75  15.8    0.735   0.094   0.031    0.001     0.578
-------------------------------------------------------------------------------------------------------------------
total            1  1.320(100.0)  1320.361   199.8     0.2 28.75  15.8    0.735   0.094   0.031    0.001     0.578

                (a)          (b)         (c)         (d)         (e)         (f)         (g)         (h)

PROC.NAME    ELAPSED      COMM.TIME  COMM.TIME   IDLE TIME  IDLE TIME  AVER.LEN      COUNT   TOTAL LEN
             TIME[sec]       [sec]  / ELAPSED      [sec]  / ELAPSED     [byte]                  [byte]

test             1.615       1.572      0.973      0.093      0.057       68.0     100000       6.5M
-------------------------------------------------------------------------------------------------------
```

図 6.7-3 FTRACE の出力例(YES 指定時)

表示される各項目の意味は以下のとおりである.
(a)～(h)以外の項目は逐次処理プログラムの FTRACE と同じである.

(a):経過時間
(b):MPI 通信の送受信に要した経過時間(MPI 手続きの実行に要した時間を含む)
(c):関数の MPI 通信の送受信に要した経過時間(MPI 手続きの実行に要した時間を含む)と, 経過時間に対する比率
(d):MPI 通信を行うまでの待ち時間, および同期待ちに要した経過時間
(e):関数の MPI 通信を行うまでの待ち時間, および同期待ちに要した経過時間と, 経過時間に対する比率
(f):MPI 通信 1 回当たりの平均通信量
(g):MPI 通信回数
(h):MPI 通信の通信量

FTRACE を用いることで, 手続き(サブルーチンや関数)単位の通信性能を知ることができる. しかしながらFTRACE では, どの MPI サブルーチンが何回呼び出されたかを知ることができない. 6.7(2)節の MPICOMMINF で得た情報と組み合わせることで, 使用された MPI サブルーチンと通信性能(通信時間, 通信量など)を分析することが可能になる. また, FTRACE の region 機能を使用することで, 特定の MPI ライブラリの通信性能情報を得ることができる. この場合, MPI_BARRIER(注 4)で同期を取ることで, 演算処理の待ち時間を含まない通信に要する時間のみを採取することができる.

（注　1）形式 0. プログラム単位名を左端に表示する.
（注　2）形式 1. プログラム単位名を右端に表示する.
（注　3）形式 2. プログラム単位名を左端に表示する. ただし, プログラム単位名の長さが 10 文字を超えるものについては, 名前の表示の直後で改行して表示する.
（注　4）MPI_BARRIER:コミュニケータ内の全プロセスで同期を取る場合に使用する. 全プロセスで MPI_BARRIER が呼ばれるまで, どのプロセスも次の処理へ進まない.

高速化推進研究活動報告　第5号

2011年9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.isc.tohoku.ac.jp