サイエンティフィック・システム研究会
Scientific Systems

HPC技術 WG
成果報告書

2010 年 5 月

サイエンティフィック・システム研究会

HPC 技術 WG

目次

1. はじめに

1.1　背景と目的

HPCの計算エンジンとなるスーパーコンピュータ・システムは近年益々PCクラスタを中心とするスカラ型並列計算機へと移行している。富士通にあってもSPARCアーキテクチャによるCPUを自主開発し、これをエンジンとするスカラ型並列計算機を世に問い、一定の成果を上げている。この10年間スカラ・プロセッサの飛躍的な性能向上のほとんどの割合を支えてきたのは動作周波数の向上であり、並列数増加に伴うOSやコンパイラ、ミドルウェアの機能・性能の向上である。一方、動作周波数向上は飽和に達しつつあり、単純なスカラ処理だけでは科学技術計算で必要とされている高速演算処理の達成が困難となりつつある。このためスカラ・プロセッサにもさまざまな工夫が取り込まれるようになってきている。また分野によってはベクトル・アーキテクチャに対する根強い需要があるのも確かであり、何がしかの“融合”が期待されているところである。

このような状況のなか、我が国においてもポスト・地球シミュレータ計画と位置付けられる「次世代スーパーコンピュータ計画」が開始されたことは、一時ほどの活況を呈さなくなっているHPC社会―国産スーパーコンピュータメーカのみならず利用者コミュニティも含めて―に活力を与えるものと大きな期待が懸けられている。この計画では富士通もスカラ並列計算機部分の開発を担って参加しており、プロセッサ開発技術の維持・向上・涵養、独自プロセッサを持つことによるソフトウェア開発力に関する同様の効果を発揮させることが期待される。

こうしたことからSS研にあっても、HPC技術・HPC社会の新しい動きに呼応して、HPC技術全般に関わることができるワーキング・グループ活動に取り組むべきではないかとして「HPC技術WG」を立ち上げることとした。また1システムとしての計算機処理能力の向上がハードウェアに依存するだけでなく、ソフトウェア全般に依存する状況が従来に増して大きくなっていることから、これまでのＷＧ活動では比較的的を絞った形のテーマで取り組んできたが、今回のWGでは、とりあえず従来通りの活動形態として「性能評価」活動に取り組むが、これ以外にもＷＧ活動を進めていくなかで関心あるテーマが出てきたら、サブＷＧ活動として随時取り組むことができる体制とした。本報告は平成19年から3カ年にわたるWG活動の内容をまとめたものである。

1.2. 活動経緯

1.2.1. 活動メンバ（全体）

	氏名	機関名	就任年度		
			2007	2008	2009
担当幹事	石井 克哉	名古屋大学	○	○	○
推進委員	福田 正大 ※	計算科学振興財団（前・宇宙航空研究開発機構、個人会員）	○	○	○
	藤田 直行	宇宙航空研究開発機構	－	○	○
	高木 亮治	宇宙航空研究開発機構	－	○	○
	岩下 武史	京都大学	□	○	○
	牧野 淳一郎	国立天文台	□	○	○
	吉岡 諭	東京海洋大学	○	○	○
	平野 靖	山口大学（前・名古屋大学）	○	○	○
	吉田 啓之	日本原子力研究開発機構	○	○	○
	中村 純	広島大学	□	○	○
	南部 伸孝	上智大学（前・九州大学）	○	○	○
	市川 真一 ※	富士通(株)ﾃｸﾆｶﾙｺﾝﾋﾟｭｰﾃｨﾝｸﾞｿﾘｭｰｼｮﾝ事業本部	○	○	○
	清水 俊幸	富士通(株)次世代ﾃｸﾆｶﾙｺﾝﾋﾟｭｰﾃｨﾝｸﾞ開発本部	－	○	○
	青木 正樹	富士通(株)次世代ﾃｸﾆｶﾙｺﾝﾋﾟｭｰﾃｨﾝｸﾞ開発本部	○	○	○
	山中 栄次	富士通(株)次世代ﾃｸﾆｶﾙｺﾝﾋﾟｭｰﾃｨﾝｸﾞ開発本部	－	－	○
	杉崎 由典	富士通(株)次世代ﾃｸﾆｶﾙｺﾝﾋﾟｭｰﾃｨﾝｸﾞ開発本部	－	□	○
	内藤 俊也	富士通(株)次世代ﾃｸﾆｶﾙｺﾝﾋﾟｭｰﾃｨﾝｸﾞ開発本部	□	□	○
	軽部 行洋	富士通(株)ﾃｸﾆｶﾙｺﾝﾋﾟｭｰﾃｨﾝｸﾞｿﾘｭｰｼｮﾝ事業本部	－	○	○
	本間 節夫	富士通(株)ﾃｸﾆｶﾙｺﾝﾋﾟｭｰﾃｨﾝｸﾞｿﾘｭｰｼｮﾝ事業本部	－	○	○
	天野 一英	富士通(株)文教ソリューション事業本部	□	○	○
	鈴木 清文	富士通(株)ミドルウェア事業部本部	－	□	○
	久門 耕一	(株)富士通研究所	○	○	○
協力メンバ	長屋 忠男	富士通(株)次世代ﾃｸﾆｶﾙｺﾝﾋﾟｭｰﾃｨﾝｸﾞ開発本部	－	□	－
	宇野 俊司	富士通(株)次世代ﾃｸﾆｶﾙｺﾝﾋﾟｭｰﾃｨﾝｸﾞ開発本部	－	□	□
	石附 茂	富士通(株)ﾃｸﾆｶﾙｺﾝﾋﾟｭｰﾃｨﾝｸﾞｿﾘｭｰｼｮﾝ事業本部	□	□	□
	稲荷 智英	富士通(株)ﾃｸﾆｶﾙｺﾝﾋﾟｭｰﾃｨﾝｸﾞｿﾘｭｰｼｮﾝ事業本部	□	－	□
	吉田 真和	富士通(株)文教ソリューション事業本部	□	□	－
	松井 泰敏	富士通(株)文教ソリューション事業本部	□	□	－
	志田 直之	富士通(株)ミドルウェア事業本部	－	□	□

※：まとめ役。　○：委員として参加。　□：協力メンバとして参加（協力メンバは会合出席 3 回以上を掲載）

1.2.2. 活動メンバ（サブ WG）

■ 性能評価サブ WG メンバ

吉岡 諭 ※	東京海洋大学
中村 純	広島大学
青木 正樹	富士通(株)次世代ﾃｸﾆｶﾙｺﾝﾋﾟｭｰﾃｨﾝｸﾞ開発本部
杉崎 由典	富士通(株)次世代ﾃｸﾆｶﾙｺﾝﾋﾟｭｰﾃｨﾝｸﾞ開発本部

■ プログラミングモデルサブ WG

平野 靖 ※	山口大学（前・名古屋大学）
高木 亮治	宇宙航空研究開発機構
牧野 淳一郎	国立天文台
鈴木 清文	富士通(株)ミドルウェア事業部本部
久門 耕一	(株)富士通研究所

■ ベンチマークジョブサブ WG

南部 伸孝 ※	上智大学（前・九州大学）
軽部 行洋	富士通(株)ﾃｸﾆｶﾙｺﾝﾋﾟｭｰﾃｨﾝｸﾞｿﾘｭｰｼｮﾝ事業本部

※：まとめ役。

1.2.3. WG 会合の活動経緯

各会合の概要を下表に示す。

会合	日時	場所	活動内容
第 1 回	2007 年 7 月 10 日(火) 14:30～17:45	富士通本社	・会員報告（各社マシンの性能比較など）とその議論/検討 ・富士通報告（スカラ性能、自動並列化）とその議論/検討 ・今後の進め方、他に取り上げるべきテーマの検討
第 2 回	2007 年 10 月 23 日(火) 14:30～17:30	富士通本社	・会員報告（ベンチマークテスト報告など）とその議論/検討 ・富士通報告（スカラ・スレッド性能、メモリレイテンシ測定、各種メモリのアーキテクチャ）とその議論/検討 ・今後の進め方、他に取り上げるべきテーマの検討
第 3 回	2008 年 1 月 22 日(火) 14:00～17:30	富士通本社	・会員報告（C 言語と Fortran の比較など）とその議論/検討 ・富士通報告（スカラ・スレッド性能、姫野ベンチ、LMbench 測定方法）とその議論/検討 ・新規サブテーマ案（自動並列化、ベンチマーク作り）の提案
第 4 回	2008 年 4 月 4 日(金) 14:00～17:30	富士通本社	・会員報告（分子動力学プログラムによる測定評価、C 言語と Fortran、QCD プログラムによる Fortran90 評価）とその議論/検討 ・富士通報告（姫野ベンチの特徴、最新サーバアーキテクチャ）とその議論 ・サブテーマ「自動並列化」「ベンチマーク作り」「性能評価」のすすめ方の検討 ・新規サブテーマ案（MPI など）の検討
第 5 回	2008 年 5 月 23(金) 14:00～17:45	富士通本社	【性能評価】: Intel クアッドコア CPU でのベンチマークの報告、書式付 I/O 性能の報告 【言語比較】: C++逐次性能の調査結果、C OpenMP 性能の調査結果 【MPI】: MPI 性能比較などの報告、PSI プロジェクト(高機能スイッチ)、インターコネクト概略の報告 ・サブテーマ案「ストレージ IO」の活動提案とその検討
第 6 回	2008 年 6 月 26 日(木) 14:00～17:30	Platform Solution Center	【性能評価】: 並列効果の低下とラージページの関係の報告、Intel クアッドコア CPU でのベンチマークの報告、SUN FIRE X4600 におけるコンパイラの比較、モジュール内関数とサブルーチンの性能差 【MPI】: MPI 性能測定結果、MPI でサポートする通信のアルゴリズムなどの報告 【ストレージ IO】: 進め方の検討
第 7 回	2008 年 9 月 5 日(金) 14:00～17:30	富士通本社	【性能評価】: Intel クアッドコア CPU でのベンチマークの報告 【MPI】: IO アーキテクチャ、データモデル、MPI-IO、性能測定などの報告 ・新規活動テーマ案「並列処理人口の拡大」「大規模ストレージ」の検討
第 8 回	2008 年 11 月 7 日(金) 14:00～17:00	富士通本社	【性能評価】: 書式付 IO 性能：他社マシンでの測定結果、スレッドプログラムの並列性能の報告、Fortran90&95 言語仕様の性能面への影響の報告、FX-1 の LINPACK 性能の報告 【MPI】: MPI-IO 化候補アプリ:UPACSIO 分析の報告 ・過去の WG で作成した並列化ガイドの確認と、今後のガイド整備の確認 ・ベンチマークジョブサブ WG の活動内容の検討
第 9 回	2009 年 1 月 28 日(水) 14:00～17:15	富士通本社	【性能評価】: 書式付 IO 性能の追加報告、Intel クアッドコア CPU でのベンチマークのまとめ報告、Fortran90&95 言語仕様の性能面への影響について追加報告、FX1 の LINPACK 性能と OS ジッタの報告 【MPI】: 机上検証：MPI-IO 性能と、ストライドデータと MPI-IO 性能との関連の報告、実検証：UPACS の MPI-IO 評価予定の報告 ・ベンチマークジョブサブ WG の進め方、検討/協議事項の検討
第 10 回	2009 年 4 月 17 日(金) 14:00～17:30	富士通本社	【性能評価】: 書式付 IO 性能での浮動小数点処理の性能について、FX1 の LINPACK 性能の報告 【MPI】: 机上検証：まとめ資料のレビュー、実検証：UPACS の MPI-IO 分析の報告 ・ベンチマークジョブサブ WG の状況報告 ・今後の進め方の検討、活動テーマの整理とサブ WG の設定

第 11 回	2009 年 7 月 17 日(金) 14:00～17:10	富士通本社	【性能評価サブ WG】: サブ WG 活動の整理と進め方検討、FX1 の LINPACK 性能の追加報告、最新 TOP500 状況報告 【ﾌﾟﾛｸﾞﾗﾐﾝｸﾞﾓﾃﾞﾙｻﾌﾞ WG】: サブ WG 活動の整理と進め方検討、Fortran90 基礎性能評価報告、MPI 論理的検証報告 【ﾍﾞﾝﾁﾏｰｸｼﾞｮﾌﾞｻﾌﾞ WG】:SS 研ベンチマーク方針の検討､試作 BMT セットの実行結果報告
第 12 回	2009 年 10 月 23 日(金) 14:00～18:00	富士通本社	【性能評価サブ WG】: 成果報告書に向けた整理、PA イベント情報活用チュートリアル報告、MPI 性能検証報告 【ﾌﾟﾛｸﾞﾗﾐﾝｸﾞﾓﾃﾞﾙｻﾌﾞ WG】: 成果報告書に向けた整理、Fortran と C 言語の速度比較報告、Fortran９０基礎性能報告、MPI 論理的検証報告、MPI 実アプリ検証報告、プログラミング指針報告 ・成果報告書の概要検討
第 13 回	2010 年 1 月 6 日(水) 14:00～17:15	富士通本社	【性能評価サブ WG】: Intel クアッドコア CPU でのベンチマークの報告、MPI 性能検証報告 【ﾌﾟﾛｸﾞﾗﾐﾝｸﾞﾓﾃﾞﾙｻﾌﾞ WG】: Fortran と C 言語の速度比較報告、プログラミング指針追加報告 【ﾍﾞﾝﾁﾏｰｸｼﾞｮﾌﾞｻﾌﾞ WG】: 成果報告書案報告 ・成果報告書の詳細検討
第 14 回	2010 年 3 月 11 日(木) 14:00～17:30	富士通本社	・成果報告書のレビュー ・後継/新設 WG の検討

2. 性能評価サブ WG 報告

2.1. サブ WG まとめ

性能評価サブ WG まとめ役
東京海洋大学　　吉岡　諭

このサブ WG では、HPC のハード面の性能について、各社マシンの基礎性能の比較や、実アプリでの比較、自動並列化能力の比較、書式付 IO 性能の比較などを行う、ということがテーマになっている。HPC 技術 WG が発足した 2007 年には、PC の CPU はデュアルコアが主流となり始めていて、クアッドコアの CPU はまだ少なかった。現在ではクアッドコアの CPU がノート PC にも搭載されるようになってきており、一方ハイエンドでは 6 あるいは 8 コアの CPU が出荷され始めている。そういう意味で当 WG の活動期間は CPU のマルチコア化の時代だったということになる。そのようなマルチコア CPU のクラスタがこれからの HPC の一つの方向性だとして、それではそのような計算機の性能はどのようなものなのか、またどのような並列化手法が良いのかを検討することが、このサブ WG の目的だったと言えよう。今後 HPC 向けの計算機において、CPU がさらにコア数を増やして、メニーコア化に向かうのか、または GPU のようなアクセラレータを備えたものが主流になるのか、まだ分からないところではあるが、ここでは現時点での計算機及び並列化手法の性能評価の結果を報告する。

性能評価は、既存の、あるいは新たに作成した種々のベンチマークプログラムを用いて行われた。対象となったハードウェアは CPU や構成等様々であり、また、コンパイラ、並列化手法（自動並列化、OpenMP、MPI)も様々である。よって、性能評価の結果は、たくさんの座標軸がある空間でのものとなっていて、そこからひとつの結論を導き出すのは難しいかもしれないが、これだけ様々なベンチマークを実行するのは個人ではなかなか出来ないことであり、今後 HPC 向け計算機を構築する際の一助になれば幸いである。また、必ずしも性能評価そのものではないが、性能向上を図るために役立つと思われるいくつかの報告もあるので、参考にして頂きたい。

2.2. 基礎性能

2.2.1. スカラおよびスレッド並列性能評価

富士通株式会社　　青木 正樹

ここでは、スカラ（1CPU コア）性能およびスレッド並列性能の評価結果について報告する。

1. 性能評価のポイント

スカラおよびスレッド並列性能の評価分析のポイントは、以下の 4 点。

・コンパイラの最適化/並列化の能力
・CPU の演算器性能
・メモリアクセス性能（バンド幅、レイテンシ）
・スレッド並列時のオーバヘッド

各々の性能要件は、アプリコードがどのようにコーディングされているかにより異なるが、定性的に上記観点を評価分析すれば、おおよその性能評価/分析は可能である。

2. 評価対象 CPU および評価対象コード

今回の評価に用いた CPU を以下に示す。

大分類	分類
SPARC 系	SPARC64V　1 チップ/1 コア 1.3GHz
	SPARC64Ⅵ　1 チップ/2 コア 2.28GHz
	SPARC64Ⅶ　1 チップ/4 コア　2.5GHz
IPF	Montecito　1 チップ/2 コア　1.6GHz
X86 系	Xeon/Woodcrest　1 チップ/2 コア　3.0GHz
	Xeon/Clovertown　1 チップ/4 コア　2.6GHz
	Opteron/Barcelona　1 チップ/4 コア　2.3GHz
VPP	VPP5000　1PE　9.6GFLOPS

今回の評価に用いたコンパイラを以下に示す。

- 富士通製：Parallelnavi Language Package for Linux および Soraris V3 系
- 他社コンパイラ：Intel コンパイラ 10 系、SUN Studio 11

今回の評価に用いたコードを以下に示す。

コード	目的	備考
Netlib/vectord	コンパイラの自動並列化能力	公開
実コード 254 本（高ｺｽﾄﾙｰﾌﾟ）	コンパイラの自動並列化能力	非公開
STREAM Benchmark	メモリ性能（バンド幅）	公開
LMBench	メモリ性能（レイテンシ）	公開
EuroBen Benchmark	カーネルループの演算性能	公開
OpenMP Micro Benchmarks	OpenMP オーバーヘッド時間	公開
姫野ベンチ	アプリコード評価	公開

各コードの詳細は以下のとおり。

【Netlib/vector】

- http://netlib.org/benchmark/vectord
- 本来は、20 年近く前に自動ベクトル化コンパイラの評価用に作成されたコード。
- しかし、ベクトル化と類似の自動並列化の評価にも利用できると判断。
- 全 135 ループからなる。

【実コード 254 本（高コストループ)】
- 富士通のコンパイラ開発部隊が保有する実コード群。
- 通常は、出荷前の性能試験に利用。
- ベクトル機の時代からの BMT コード&著名 BMT コードから構成。
- 各コードの実行コスト 90%以上のルーチンを抜き出し、翻訳テストにも利用。総ループ 7000 以上。

【STREAM Benchmark】
- http://www.cs.virginia.edu/stream/
- HPC のアプリにとり、メモリバンド幅は最重要。STREAM で公開されているのは総バンド幅
- Web には、全ての値（真の実力）が公開されていない場合がある。

STREAM（公表データ一例）

システム	CPU	CPUS	MB/s	1 コアあたりの MB/s	B/F 比
Fujitsu VPP5000	vector	1	37544.0	37544.0	3.91
IBM System p5 595	powoer5+ 2.3GHz	64	206243.0	3222.5	0.35
Fujitsu Enterprise M9000	SPARC64_VI 2.4GHz	128	227059.0	1773.9	0.18
Fujitsu PRIMEQUEST＊	Montecito 1.6GHz	64	82755.0	1293.0	0.20

＊は、未公開だが実測値。

【LMBench】（ここではメモリレイテンシに特化し説明）
- http://www.bitmover.com/lmbench/
- ポインターチェイン（次のアドレスは前のメモリアクセスが終らないと決まらない ture dependency の関係を利用）の単純コード。
- 1 プロセス/整数ロード実行。（一般的には一番近いメモリアクセスの時間）
- L1 ⇒ L2 ⇒（L3⇒）メモリのレイテンシ変移をみることができる。

【Euro Ben Benchmark】
- http://www.euroben.nl/
- 科学技術計算用コンピュータの単体 CPU 性能と MPI 並列性能を測定するためのベンチマークプログラム。
- ベンチマーク手順の合理化と規格化の促進のために、オランダ Utrecht Univ.の High Performance Computing Group（HPCG）により Working Group が 1990 年に設立され、現在は V5.0 が公開されている。OpenMP・MPI 版もあり。
- 以下の 3 つの Module からなる。
  Module1: 素性能（演算性能，メモリ性能，関数性能）
  Module2: 基本数学アルゴリズム性能（行列積，連立方程式，固有値，FFT など）
  Module3: 微分方程式,ポアソン方程式など
- 新しいチップの測定結果が公開されている。

【OpenMP　MicroBenchmarks】
- http://www2.epcc.ed.ac.uk/computing/research_activities/openmpbench/openmp_index.html
- OpenMP でサポートされている各ディレクティブのオーバヘッド時間を計測するコード
  ⇒スレッド並列時のオーバヘッド時間の素性能
  （各ベンダの実現方法にもよるが、自動並列化のオーバヘッドと考えてもよい）

3. コンパイラ評価

Vector/実コードを用い、自動並列化能力を評価した。

・並列化有無の判断は、コンパイラ出力の MSG による。

・ Euro Ben Benchmark を用い、X86 用コンパイラの逐次実行性能を評価。

■評価まとめ

・富士通コンパイラはトップランナーでなくなった。(Vector 評価より)

・しかし、実コードでは高水準。ベクトル化技術継承。(実コード評価より)

・近い将来、自動並列化能力は、自動ベクトル化同等以上となる。(予測)

・X86 コンパイラの逐次実行性能は、富士通コンパイラ対 Intel に対して凸凹。

3.1 Vector コードによる自動並列化能力評価結果（まとめ）

	コンパイラ	並列ループ数	ベクトル化比
参考	ベクトル化(VPP)	85	0.98
推定	理想の並列化コンパイラ	87	1.00
1位	SUN	72	0.83
2位	富士通	67	0.77
3位	INTEL	61	0.70
4位	PGI	10	0.10

各オプション

富士通　-Kfast,parallel_strong -Qt -Et

Sun　-fast -parallel -loopinfo -reduction

Intel　-O3 -ipo -xW -parallel -par-threshold0 -par-report3 -vec-report5

PGI　-fastsse -Mconcur=innermost -Minfo

3.2 Vector コードによる自動並列化能力評価結果（ループ毎詳細抜粋）

	VPP	FJ	SUN	PGI	INTEL	理想(推定)
並列化個数	85	87	72	10	61	87
ループ番号						
S111	○	○	○	×	○	○
S112	○	×	×	×	×	×
S113	○	○	○	×	○	○
S114	○	○	×	○	×	○
S115	○	○	×	×	○	○
S116	○	×	○	×	×	○
S118	×	○	×	×	○	○
S119	○	○	×	×	○	○
S121	○	×	×	×	×	×
S122	○	○	○	×	×	○
S123	×	×	×	×	×	×
S124	×	×	×	×	○	×
S125	○	○	×	×	○	○
S126	×	×	○	×	×	○

※FJ は富士通を表す。以降同様。

【Vector コード：FJ のみ自動並列化】

最大/最小位置検索

```
S331
F  並列化可

     3950    1                DO 1 NL = 1,NTIMES
     3951    1                J  = -1
     3952    2  pp u           DO 10 I = 1,N
     3953    2   p u              IF(A(I) .LT. 0) J = I
     3954    2   p          10  CONTINUE
     3955    1                CHKSUM = DBLE(J)
     3956    1                CALL DUMMY(LD,N,A,B,C,D,E,AA,BB,CC,CHKSUM)
     3957    1              1   CONTINUE
```

【Vector コード：FJ 自動並列化不可】

複雑(？)な配列添え字

```
S174
F  並列化不可

    2273    1              DO 1 NL = 1,2*NTIMES
    2274    2  s u          DO 10 I= 1, N/2
    2275    2  s u             A(I) = A(I+N/2) + B(I)
    2276    2  s u      10   CONTINUE
    2277    1              CALL DUMMY(LD,N,A,B,C,D,E,AA,BB,CC,1.D0)
    2278    1          1    CONTINUE
```

【Vector コード：ベクトル化のみ可能】

ループからの飛び出し

```
S332
VPPのみ 並列化可

     3977    1                DO 1 NL = 1,NTIMES
     3978    1                INDEX = -1
     3979    1                VALUE = -1.D0
     3980    2  s u           DO 10 I = 1,N
     3981    3  s u              IF ( A(I) .GT. T ) THEN
     3982    3    u                INDEX = I
     3983    3    u                VALUE = A(I)
     3984    3    u                GOTO 20
     3985    3  s u              ENDIF
     3986    2  s u         10 CONTINUE
     3987    1              20 CONTINUE
     3988    1                CHKSUM = VALUE + DBLE(INDEX)
     3989    1                CALL DUMMY(LD,N,A,B,C,D,E,AA,BB,CC,CHKSUM)
     3990    1              1  CONTINUE
```

収集・拡散ループ

```
S342
VPPのみ 並列化可

     4039    1              DO 1 NL = 1,NTIMES
     4040    1              J = 0
     4041    2  s u         DO 10 I = 1,N
     4042    3  p u            IF(A(I) .GT. 0.D0)THEN
     4043    3  m u               J    = J + 1
     4044    3  s u               A(I) = B(J)
     4045    3  p u            ENDIF
     4046    2  p u        10  CONTINUE
     4047    1              CALL DUMMY(LD,N,A,B,C,D,E,AA,BB,CC,1.D0)
     4048    1            1   CONTINUE
```

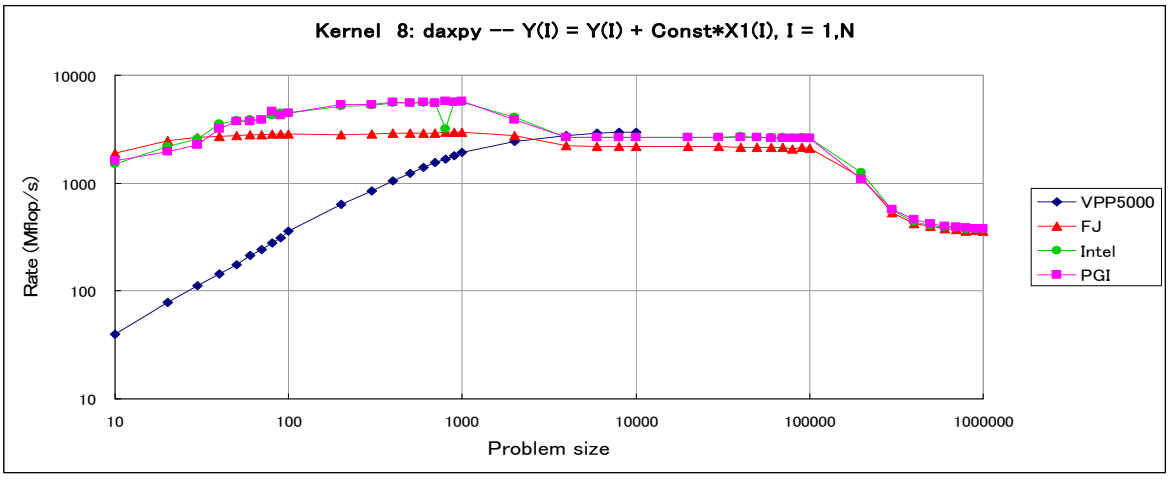
3.3 実コードによる自動並列化能力評価結果（まとめ）

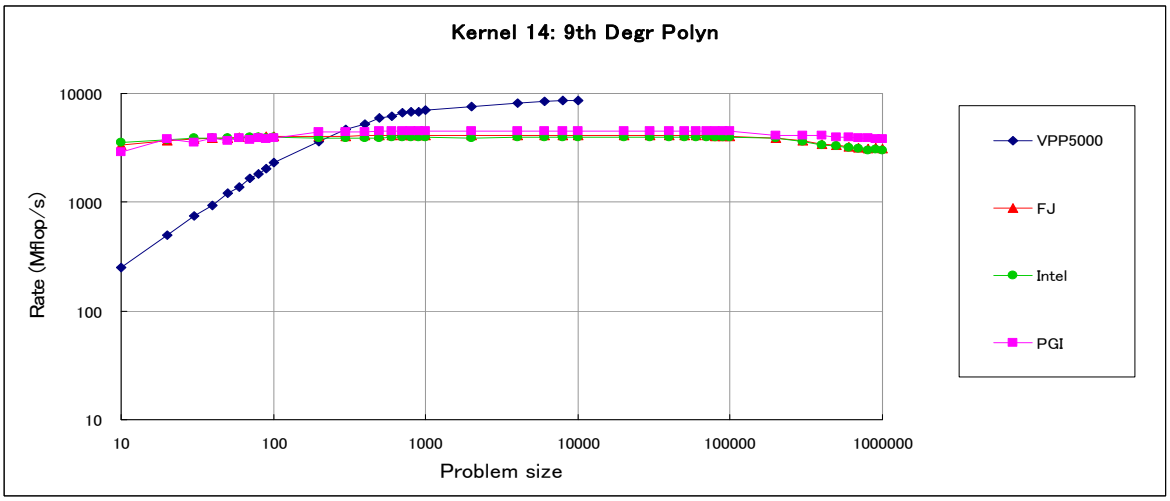
	コンパイラ	並列ループ数	ベクトル化比
参考	ベクトル化(VPP)	5590	1.00
1位	富士通	5020	0.90
2位	Intel	4643	0.83
3位	SUN	3015	0.54
4位	PGI	1130	0.20

富士通のコンパイラ開発評価に使用しているコード群であるため、富士通コンパイラ優位なコードである可能性十分あり。今後、詳細分析を行なう必要がある。

3.4 コンパイラ　スカラ性能比較

Euro Ben Benchmark を使用し、IA コンパイラベンダの各コンパイラのスカラ性能を評価した。
実行マシン：Woodcrest 3.0GHz





基本ループの最適化能力は、各ベンダほぼ同等。

4. CPU の演算性能

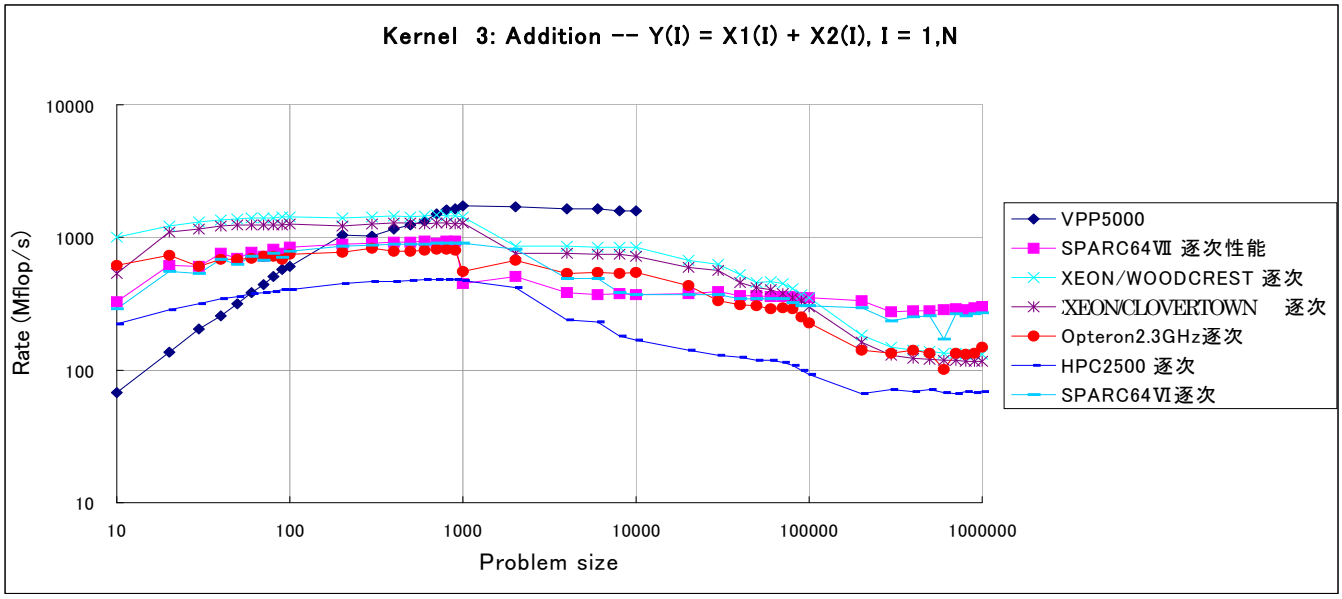
Euro Ben Benchmark を用い、各 CPU のスカラ（1CPU コア）の性能を実測。カーネルループは代表的な 4 種を評価。

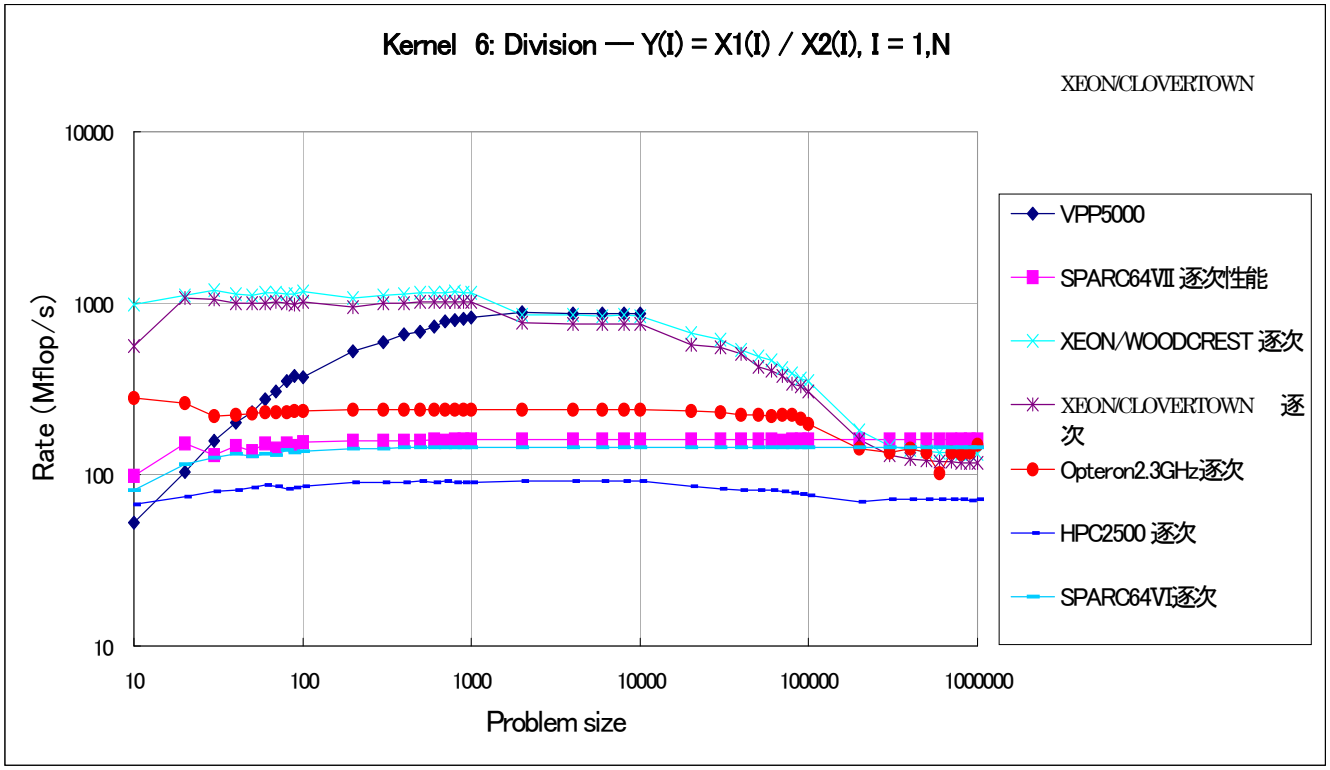
■評価まとめ

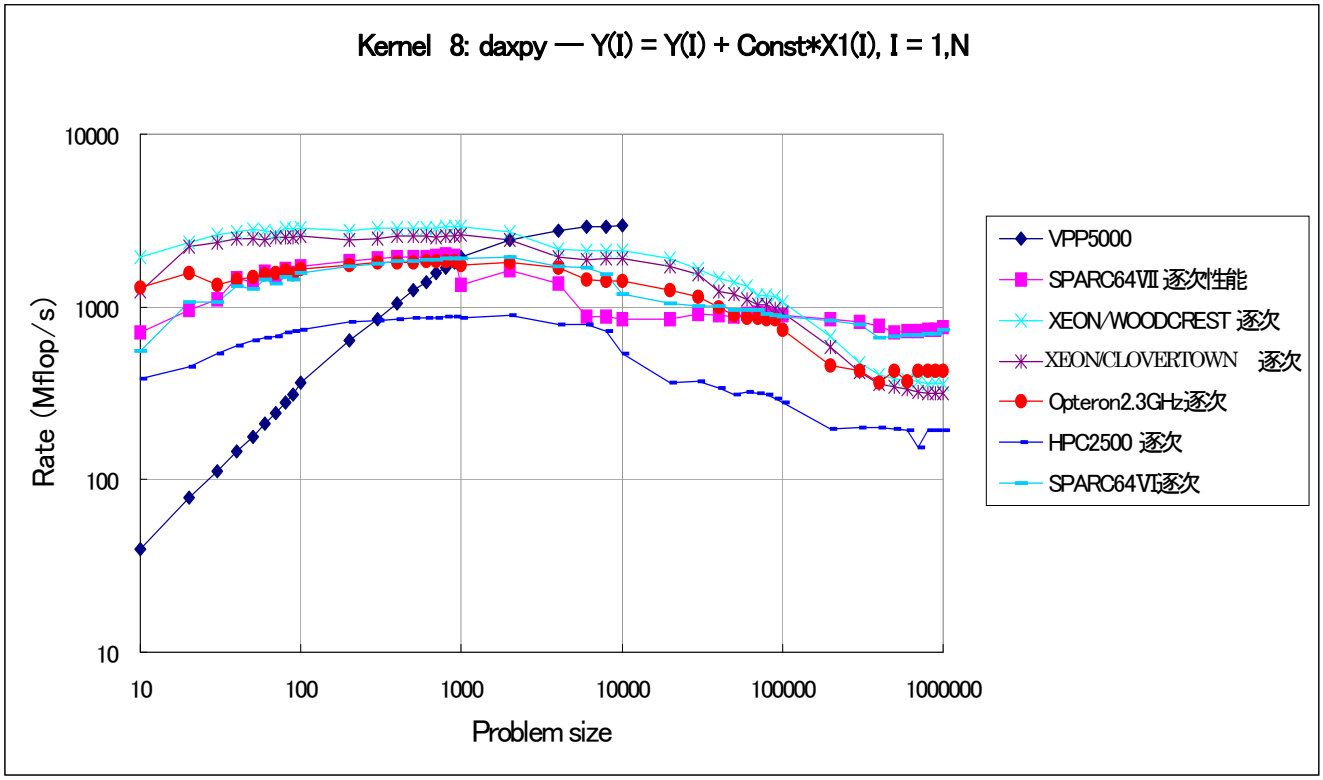
・データが L1 上にある場合、Xeon 系チップが速い。特に DIV は、同一値が連続した場合は特殊ルートで非常に高速。（Reg1=reg2/reg3--->reg4=reg2/reg1 の場合）

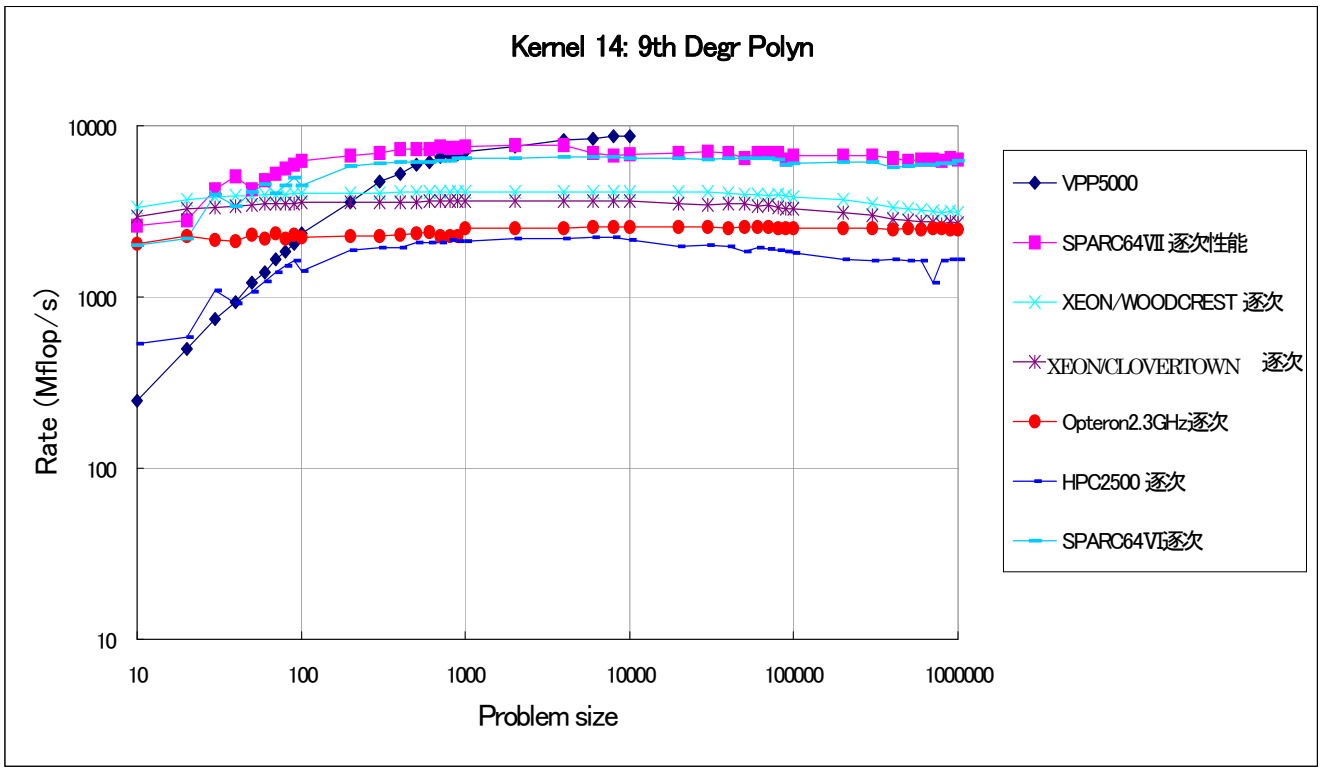
・SPARC(2.28GHz)は、演算が多いと速い。

・ピーク性能比：ベクトルはほぼ計算通り。スカラは低い。（効率悪い）






Kernel 8: daxpy — Y(I) = Y(I) + Const*X1(I), I = 1,N
Rate (Mflop/s)
10000
1000
100
10
10
100
1000
10000
100000
1000000
Problem size
VPP5000
SPARC64VII 逐次性能
XEON/WOODCREST 逐次
XEON/CLOVERTOWN 逐次
Opteron2.3GHz逐次
HPC2500 逐次
SPARC64VI逐次


Kernel 14: 9th Degr Polyn
Rate (Mflop/s)
10000
1000
100
10
10
100
1000
10000
100000
1000000
Problem size
VPP5000
SPARC64VII 逐次性能
XEON/WOODCREST 逐次
XEON/CLOVERTOWN 逐次
Opteron2.3GHz逐次
HPC2500 逐次
SPARC64VI逐次

5. メモリアクセス性能

STREAM によるバンド幅の評価、LMBench によるレイテンシ評価を行なった。

■評価まとめ

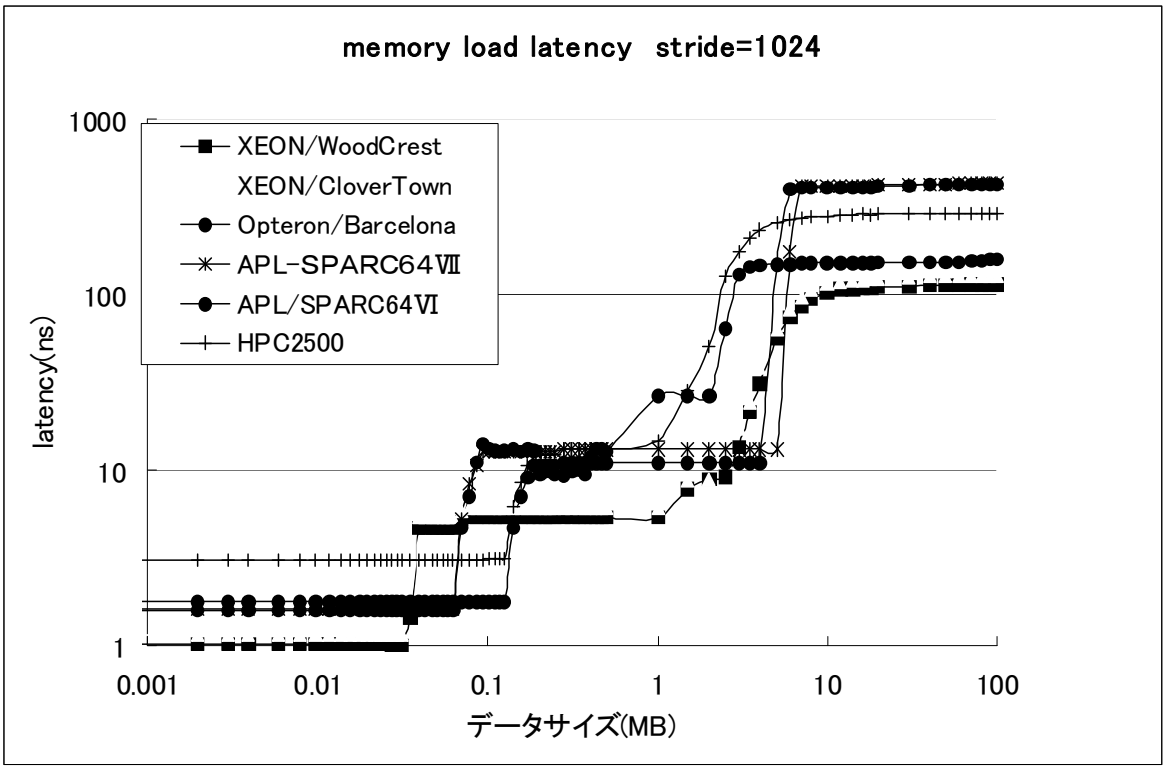
小規模 SMP ノードはレイテンシが良い。

【バンド幅】

システム	ノードあたりのチップ数	ノード全体のバンド幅	1チップのバンド幅
VPP5000	1	37.5 GB/s	37.5 GB/s
IBM P575	8	86.1 GB/s	10.4 GB/s (注 1)
HPC2500	128	60 GB/s	1.7 GB/s
PQ580/IPF	32	77.1 GB/s	5.4 GB/s
APL/SPARC64VI	32	133.4 GB/s	7.6 GB/s
PG/Woodcrest	2	5.4 GB/s	3.3 GB/s
Opteron/Barcelona	16	16.0 GB/s	5.9 GB/s

(注 1)公表値。他は今回実測。

【レイテンシ】



6. スレッド並列のオーバヘッド（素性能）評価

OpenMP MICRO Bench を用い代表的な 4 つのディレクティブ（スレッド並列のオーバヘッド時間）を評価した。

■評価まとめ

・SPARC64 系 CPU は、ハードウェアバリア機構および共用キャッシュ機構の効果で高速。
・他は、メモリ(キャッシュ)レイテンシとキャッシュ機構に依存。

	SPARC64Ⅶ	SPARC64Ⅵ (2.28GHz)		Opteron (1.9GHz)		WoodCrest (3.00GHz)		CloverTown (2.66GHz)	HPC2500 (1.3 GHz)
	4 並列	2 並列	4 並列	4 並列 (FJ)	4 並列 (Intel)	2 並列	4 並列	4 並列	4 並列
PARALLELDO	0.30	3.90	9.61	1.93	1.89	0.72	2.15	2.20	2.05
DO	0.10	1.80	6.72	1.08	1.05	0.34	0.79	0.87	0.26
バリア	0.10	1.82	6.69	1.07	1.00	0.36	0.75	0.87	0.27
REDUCTION	0.50	4.16	17.46	3.29	3.35	1.20	3.17	3.35	4.43

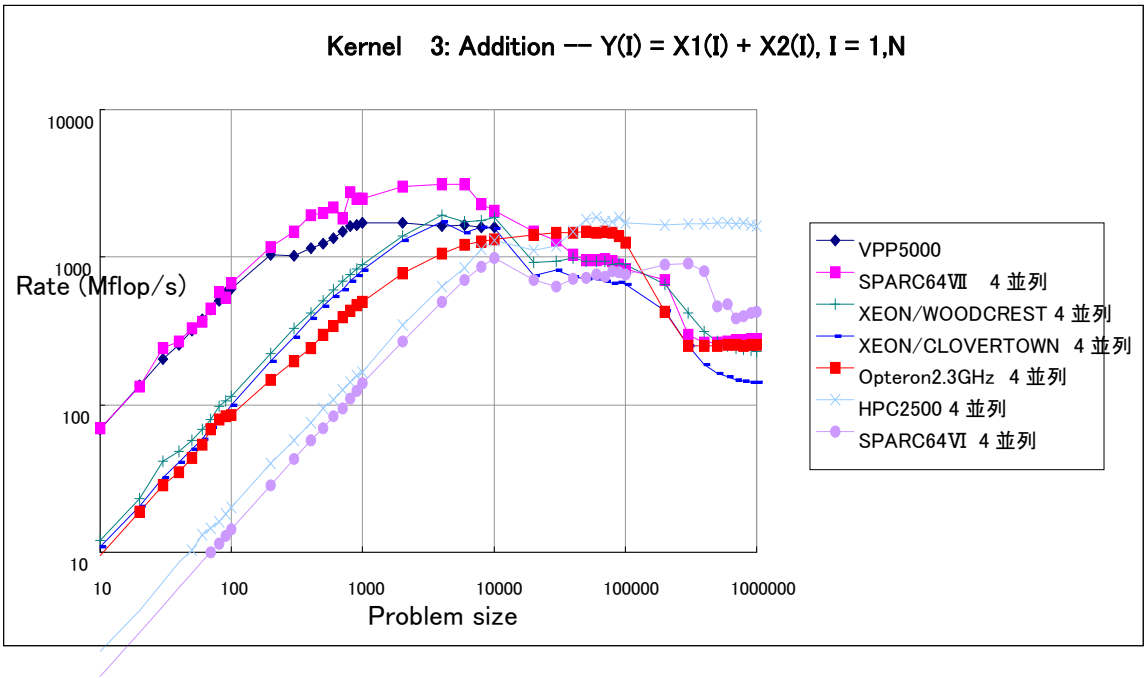
単位：μ秒

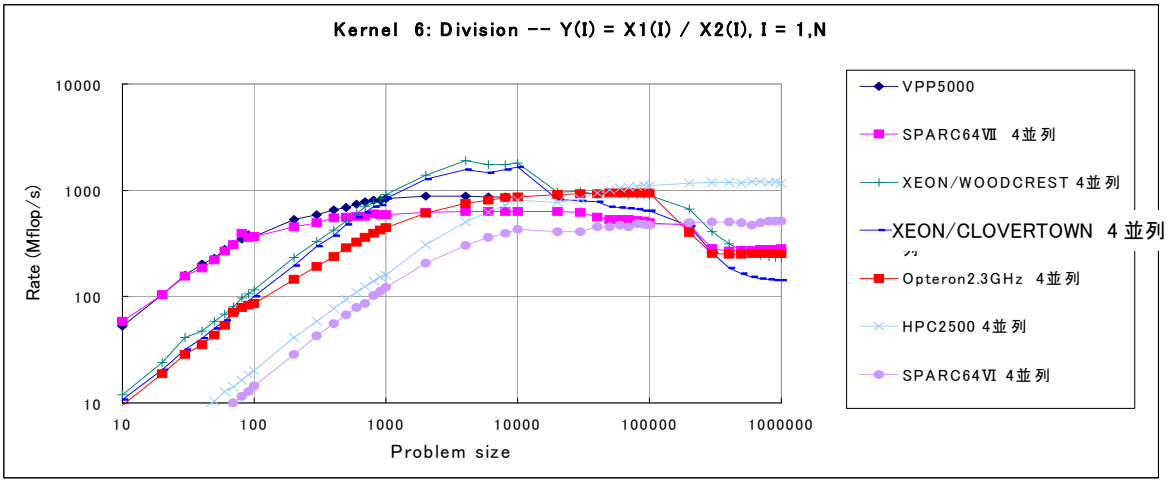
7. スレッド並列の性能評価

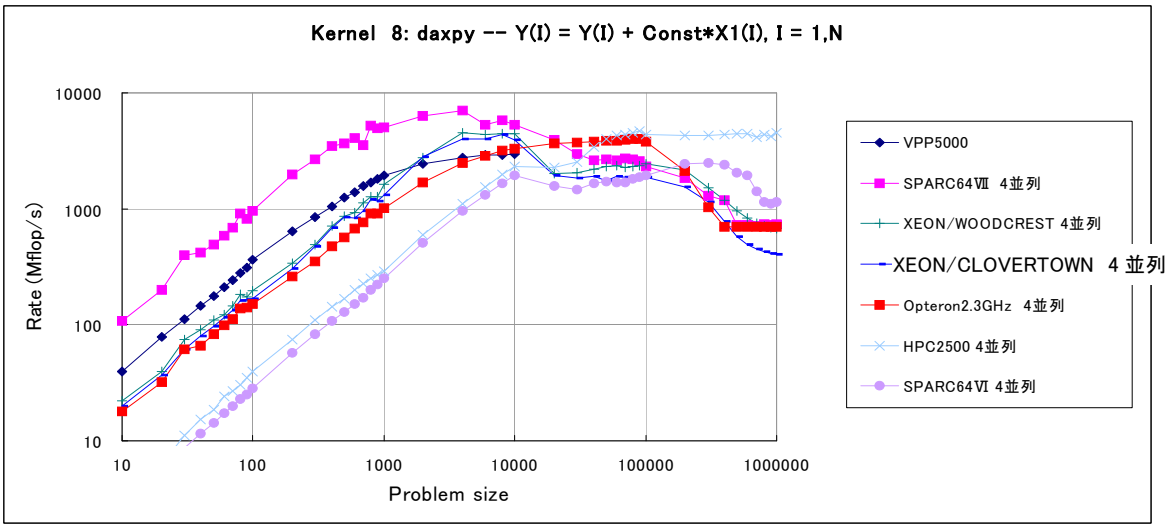
各 CPU の 4 スレッド実行時のカーネルループ実行時間評価。Euro Ben Benchmark を用い並列性能を実測。カーネルループは、代表的な 4 種を評価。

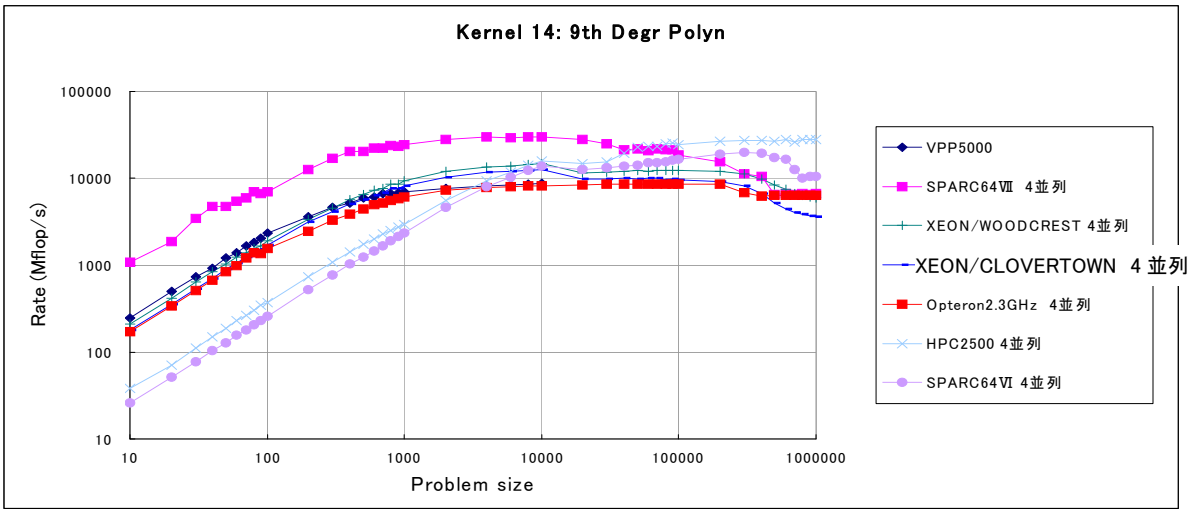
■評価まとめ

・キャッシュ機構で細粒度性能決まる。
・ベクトル並みの効率のためには、それなりのハードウェア機構必要。




Kernel 6: Division -- Y(I) = X1(I) / X2(I), I = 1,N
Rate (Mflop/s)
10000
1000
100
10
10
100
1000
10000
100000
1000000
Problem size
VPP5000
SPARC64Ⅶ 4並列
XEON/WOODCREST 4並列
XEON/CLOVERTOWN 4並列
Opteron2.3GHz 4並列
HPC2500 4並列
SPARC64Ⅵ 4並列


Kernel 8: daxpy -- Y(I) = Y(I) + Const*X1(I), I = 1,N
Rate (Mflop/s)
10000
1000
100
10
10
100
1000
10000
100000
1000000
Problem size
VPP5000
SPARC64Ⅶ 4並列
XEON/WOODCREST 4並列
XEON/CLOVERTOWN 4並列
Opteron2.3GHz 4並列
HPC2500 4並列
SPARC64Ⅵ 4並列


Kernel 14: 9th Degr Polyn
Rate (Mflop/s)
100000
10000
1000
100
10
10
100
1000
10000
100000
1000000
Problem size
VPP5000
SPARC64Ⅶ 4並列
XEON/WOODCREST 4並列
XEON/CLOVERTOWN 4並列
Opteron2.3GHz 4並列
HPC2500 4並列
SPARC64Ⅵ 4並列

8. 姫野ベンチ評価

アプリコード評価として、姫野ベンチの評価分析を実施した。

【評価したモデル】

ソースコード（Fortran90 ＋ OMP）	
サイズ	要素数
XL	(1024 × 512 × 512)
L	(512 × 256 × 256)
M	(256 × 128 × 128)
S	(128 × 64 × 64)

【カーネルコード】

```
do k=2,kmax-1
  do j=2,jmax-1
    do i=2,imax-1
      s0=a(I,J,K,1)*p(I+1,J,K) &
          +a(I,J,K,2)*p(I,J+1,K) &
          +a(I,J,K,3)*p(I,J,K+1) &
          +b(I,J,K,1)*(p(I+1,J+1,K)-p(I+1,J-1,K) &
                    -p(I-1,J+1,K)+p(I-1,J-1,K)) &
          +b(I,J,K,2)*(p(I,J+1,K+1)-p(I,J-1,K+1) &
                    -p(I,J+1,K-1)+p(I,J-1,K-1)) &
          +b(I,J,K,3)*(p(I+1,J,K+1)-p(I-1,J,K+1) &
                    -p(I+1,J,K-1)+p(I-1,J,K-1)) &
          +c(I,J,K,1)*p(I-1,J,K) &
          +c(I,J,K,2)*p(I,J-1,K) &
          +c(I,J,K,3)*p(I,J,K-1)+wrk1(I,J,K)
      ss=(s0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
      GOSA1=GOSA1+SS*SS
      wrk2(I,J,K)=p(I,J,K)+OMEGA *SS
    enddo
  enddo
enddo
```

【評価マシン諸元】

	SPARC Enterprise M9000	HPC2500	HX600	PRIMERGY RX200 S3	IPF
プロセッサ	SPARC64VII	SPARC64V	Barcelona	Woodcrest	Itanium2
周波数 (GHz)	2.5	1.3	2.3	3.0	1.6
L2 キャッシュ (MB)	6	2	0.5	4	0.25
L3 キャッシュ (MB)	-	-	2	-	9

8.1 単精度版の性能

・1CPU コア／コンパイラ毎の性能　　単位：MFLOPS

Grid size	M9000	HPC2500	HX600	PRIMERGY				IPF
				FJ	Intel	PGI	Pathscale	Intel
XS(64x32x32)	2096	513	761	2102	2312	1836	1937	2856
S(128x64x64)	2269	480	1414	1532	1386	1428	1508	1672
M(256x128x128)	2428	492	1094	1389	1321	1351	1349	1561
L(512x256x256)	2421	475	993	1260	1218	1234	1227	1515
XL(1024x512x512)	2294	472	607	-	-		-	-

・ ピーク性能比

	M9000	HPC2500	HX600	PRIMERGY	IPF
単精度演算 ピーク性能 (MFLOPS)	10000	5200	18400	24000	6400

Grid size	M9000	HPC2500	HX600	PRIMERGY				IPF
				FJ	Intel	PGI	Pathscale	Intel
XS(64x32x32)	21%	10%	4%	9%	10%	8%	8%	45%
S(128x64x64)	23%	9%	8%	6%	6%	6%	6%	26%
M(256x128x128)	24%	9%	6%	6%	6%	6%	6%	24%
L(512x256x256)	24%	9%	5%	5%	5%	5%	5%	24%
XL(1024x512x512)	23%	9%	3%	-	-		-	-

8.2 倍精度版の性能

・ 1CPU コア／コンパイラ毎の性能　　単位：MFLOPS

Grid size	M9000	HPC2500	HX600	PRIMERGY				IPF
				FJ	Intel	PGI	Pathscale	Intel
M(256x128x128)	1648	301	476	691	720	820	750	-

・ ピーク性能比

	M9000	HPC2500	HX600	PRIMERGY	IPF
倍精度演算 ピーク性能 (MFLOPS)	10000	5200	9200	12000	6400

Grid size	M9000	HPC2500	HX600	PRIMERGY				IPF
				FJ	Intel	PGI	Pathscale	Intel
M(256x128x128)	16%	6%	5%	6%	6%	7%	6%	-

8.3 多重実行性能

・ CPU チップ／コンパイラ毎の性能　　単位：MFLOPS

単精度実数型

Grid size	M9000	HPC2500	HX600	PRIMERGY				IPF	
				FJ	Intel	PGI	Pathscale	Intel	
M(256x128x128)	777	540	768	594	531	533	588	-	4 多重
M(256x128x128)	2391	549	1060	1389	1321	1351	1349	1561	1 多重

倍精度実数型

Grid size	M9000	HPC2500	HX600	PRIMERGY				IPF	
				FJ	Intel	PGI	Pathscale	Intel	
M(256x128x128)	378	295	165	282	282	326	285	-	4 多重
M(256x128x128)	1648	301	476	691	720	820	750	-	1 多重

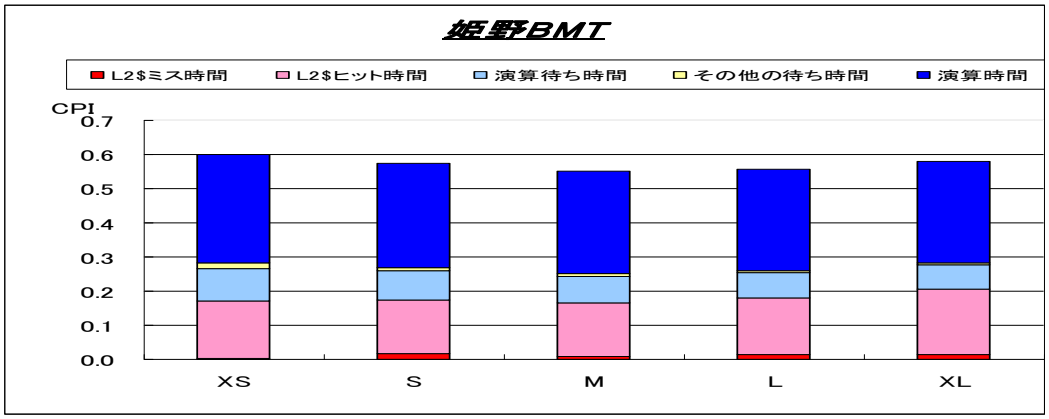
8.4 コード詳細分析

SPARC Enterprise M9000 を用いてカーネルループを詳細分析。
単精度版：1CPU コア実行

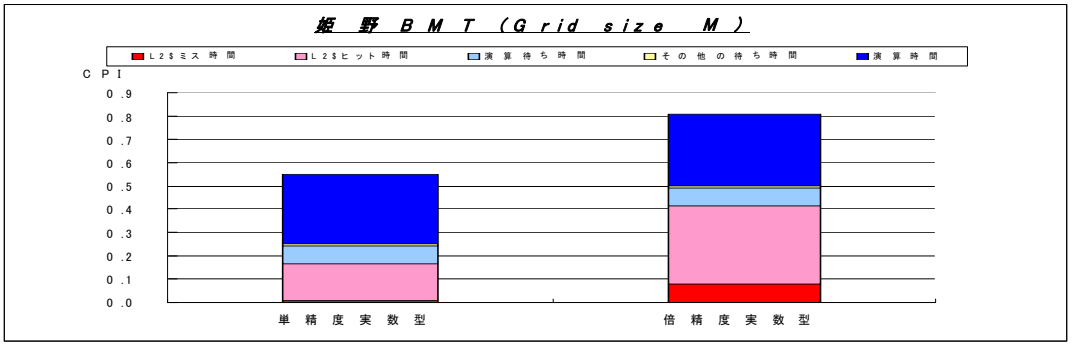
Grid size	MFLOPS	メモリ量(MB)	ﾛｰﾄﾞｽﾄｱ数(1 回あたり)	演算命令率	LD/ST 率	演算時間率	演算待ち時間率	ﾒﾓﾘｱｸｾｽ時間率			ﾒﾓﾘｽﾙｰﾌﾟｯﾄ(MB/s)
								L2$ﾋｯﾄ時間率	L2$ﾐｽ時間率	合計	
XS(64x32x32)	2096	3.5	32 (31+1)	50%	42%	53%	16%	28%	0%	29%	-
S(128x64x64)	2269	28				53%	15%	27%	3%	30%	3863
M(256x128x128)	2428	224				55%	14%	29%	2%	30%	3972
L(512x256x256)	2421	1792				53%	13%	30%	2%	32%	3883
XL(1024x512x512)	2294	14336				51%	12%	33%	2%	35%	3716

・メモリ量から、N 並列実行の場合、キャッシュに乗ってしまうため評価注意。
・メモリアクセス命令と演算命令の比率ほぼ同じ。＝アクセス性能と演算性能は両方重要。
・単精度版の場合、予想以上にメモリアクセス“のみ”がネックとなっている時間は多くない。
・2.5GFLOPS の性能のためには、バンド幅 4GB/s が必要。

PA イベント情報：各モデル毎 CPI グラフ



PA イベント情報：単精度版と倍精度版



CPI＝0.25 が MAX（1 マシンサイクルで 4 命令実行）

8.5 姫野ベンチ評価まとめ

・演算器とメモリアクセスのバランスのとれた CPU が速い
　・2.5GFLOPS の性能出すためには、4GB/s のバンド幅が必要
　　＝　5GFLOPS ならば、8GB/s
　・2.5GHz 程度のマシンサイクルならば、4GB/s でバランスしている（ﾒﾓﾘｱｸｾｽが見えない）
　　ただし、倍精度データになったら倍のバンド幅が必要
・レジスタ数の多い CPU が速い（IPF，SPARC）
　・IPF：128 個、SPARC：32 個、X86-64：16 個
　・レジスタの write-port 数は 4 個必要。HPC2500 の write-port 数は 2 個。
・キャッシュ容量は多い方が良い
　・より少ない並列数でキャッシュに乗る
　・IBM の POWER 系チップならば S モデルが L3 に乗る
　・SPARC Enterprise M9000 の L1 が倍大きかったら、性能は 1.5 倍
　　⇒今後は、高速な L1 にいかに乗せるかのコンパイラ技術も必要

9. 自動並列化

「自動並列の性能」は懐疑的に思っている人も多い。一方、「ベクトル機の自動ベクトル化は性能が出やすい」というのが万人の認識。（ベクトル化は技術が確立している）

では、自動並列化で、なぜ性能でないのか？

①スレッド並列のオーバヘッド
　バリア時間
　フォルスシュアリング（キャッシュの仕組みにより異なる）
②解析能力（コンパイラの出来）
⇒理想的には①、②を解決すれば、自動ベクトル化並みの自動並列化が可能となる。

9.1 今後の HPC ハードウェアのトレンド

・　単体性能向上の行き詰まり
　シリコンの集積度向上を命令レベルの並列処理に投入する効果減少。消費電力の制約が大きく、クロック向上は困難、キャッシュ容量の増大によりさらに電力増。
　解決手段　⇒　チップ内に複数の CPU コア

・　今後は、並列システムによる高性能実現
　CPU コア数で数千、数万以上のシステム。MPI で単純にプロセスを増すだけでは、性能向上は困難。
　解決手段　⇒　チップ内スレッド並列でプロセス並列度を抑える

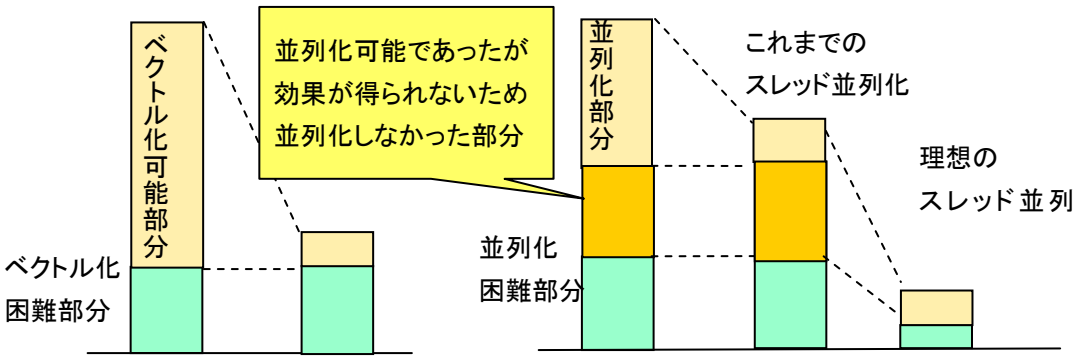
9.2 理想のスレッド並列化（①、②解決）　ベクトル化との比較

$$\text{性能向上率} = \frac{1}{(1-V) + V/\alpha}$$

V：ベクトル化率
α：ベクトル化した場合の性能向上

$$\text{性能向上率} = \frac{1}{(1-P) + P/n}$$

P：並列化率
n：並列化した場合の性能向上



9.3 チップ内（マルチコア）自動並列化で効果をだすためには

以下の要件を満足するシステムでなければならない。

・ 並列化率を高くできるか？

➔これまで、外側ループで並列化を行ってきた（疎粒度）。ベクトル化と同様に内側ループを並列化（細粒度）できれば、ベクトル並みに高い自動並列化率が可能。

・ 並列処理のオーバヘッドが十分小さくできるか？

➔内側ループを並列化すると、バリア同期の回数が非常に増加する。バリア同期の高速化が必須。また、CPU コアに接続しているキャッシュ間のデータ転送オーバヘッド削減が必要（共用キャッシュなら可能）。そして、それぞれのコアに十分なメモリバンド幅が必要。（メモリバンド幅ネックでは並列処理も意味なし）

9.4 共用キャッシュの評価

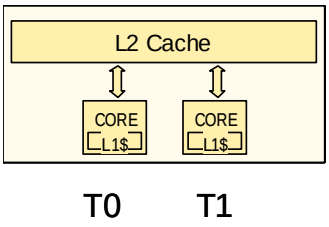
2 スレッド並列で共用キャッシュの効果を実測した。

【実測環境】

SPARC64VI（2.4GHz）

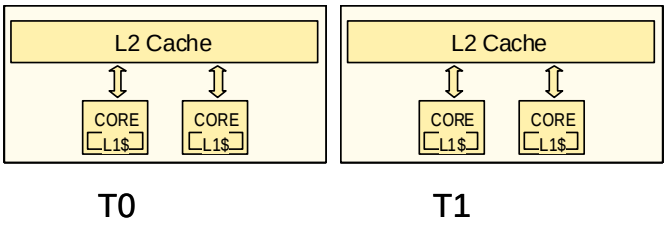
2 コア／1 チップ、L2：6MB　1 チップ内の 2 コアは L2 共用

①共用キャッシュでの実行

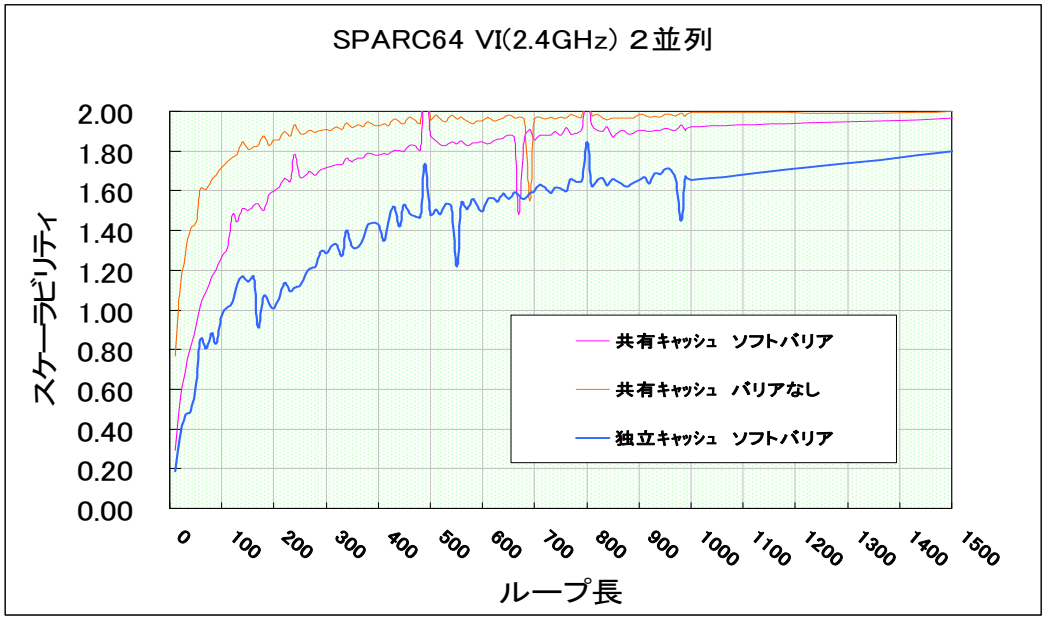


②独立キャッシュでの実行

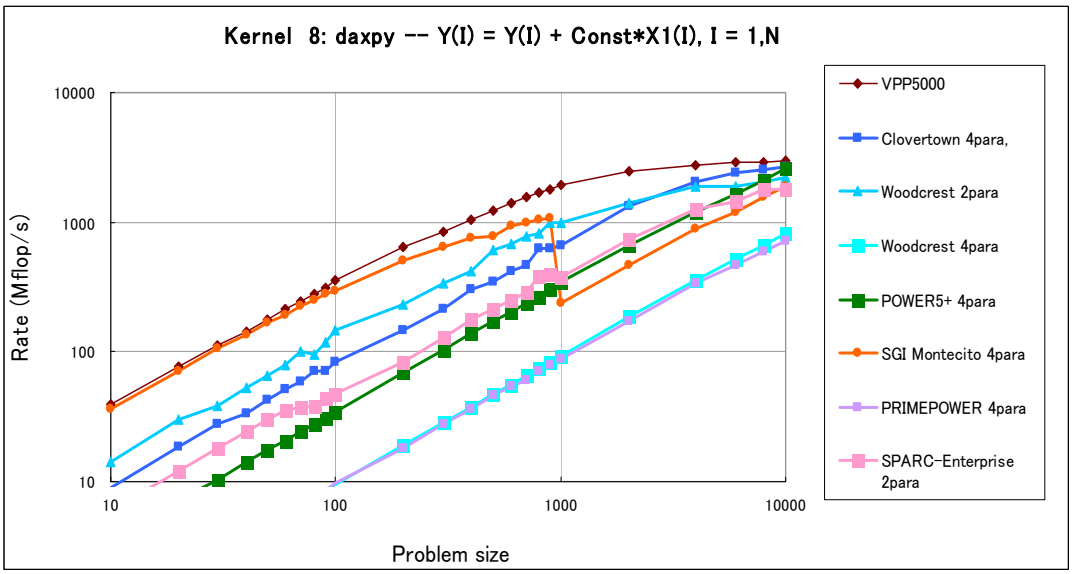
（総容量では 2 倍のキャッシュ）



【実測結果】評価コード：DAXPY



＜各種 CPU での評価＞



■評価果まとめ

スレッド並列時、コア間共用キャッシュは以下の効果がある。
①異なるスレッド（コア）間で同一キャシュラインの書き込み時に発生する false sharing の軽減
⇒細粒度のスレッド並列時、効果大

②あるスレッド（コア 1）でメモリからキャッシュに書き込んだデータが他のスレッド（コア 2）でキャッシュ上のデータとして再利用可能。
⇒ランダムアクセス時

```
do i=1,n
  a(i)=a(i)*b(l(i))
enddo
```

9.5　IA サーバー上でのコンパイラ自動並列化 比較評価

富士通製と Intel 製の 2 つのコンパイラでコンパイラの自動並列化効率を評価した。
評価の結果、コードにより凸凹あるが、逐次では Intel 製優位。並列では富士通製優位であった。

・マシン

PRIMERGY RX200 S2、　CPU：Woodcrest（2 コア/1 チップ×2）、　OS：RedHat ES4.0

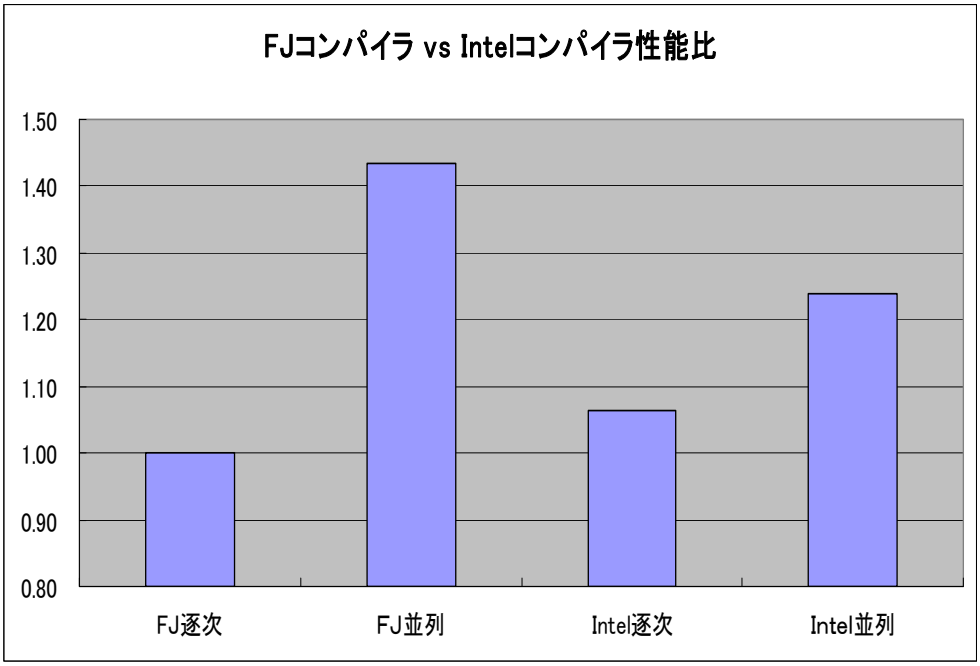
・コンパイラ

富士通コンパイラ　　options: 逐次 -Kfast –static –Kcmodel=small
並列 -Kfast,parallel -static -Kcmodel=small

Intel　Fortran Compiler 10.0.017　options: 逐次 –xP –O3 –no-prec-div -static
並列 –xP –O3 –no-prec-div -static -parallel

・評価プログラム

実コードのうち､いずれかの評価コンパイラのスケーラビリティが 1.2 倍以上となる厳選 27 本。



並列：4 スレッド並列

以上

2.2.2. MPI 性能検証

富士通株式会社　志田直之

ここでは，Open MPI および富士通 MPI を用いて，MPI 性能の評価結果について報告する。

1. 性能評価のポイント

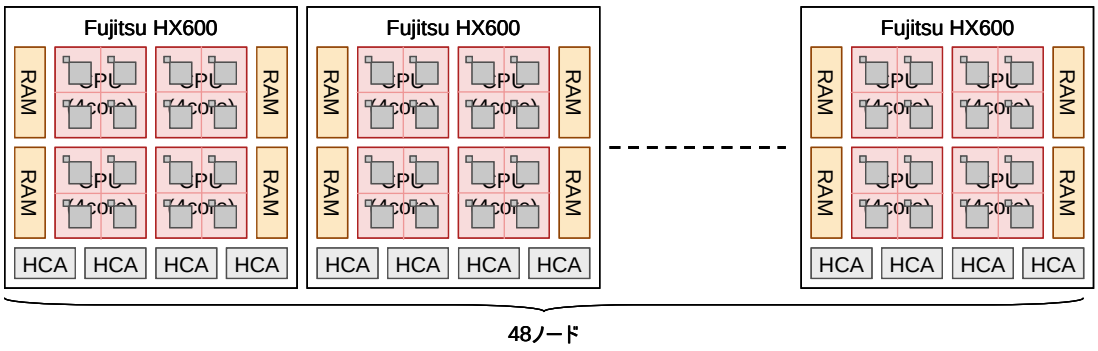
MPI の性能評価は，大きく 3 つに分けて評価を行った。

- プロセス数増加に向けた検証
- ノード内通信とノード間通信の検証
- MPI-IO 性能検証
  - 連続データ転送
  - ストライド転送

2. プロセス数増加に向けた検証

評価に用いたシステムを以下に示す。

CPU	Quad-Core AMD Opteron™ Processor 8354 2.2GHz 4CPU (16core) / node
RAM	16GB /node
Interconnect	ConnectX DDR HCA * 4
OS	Linux version 2.6.18-92.1.22.el5
Topology	FAT Tree



今回の評価に用いたコードと MPI を以下に示す。

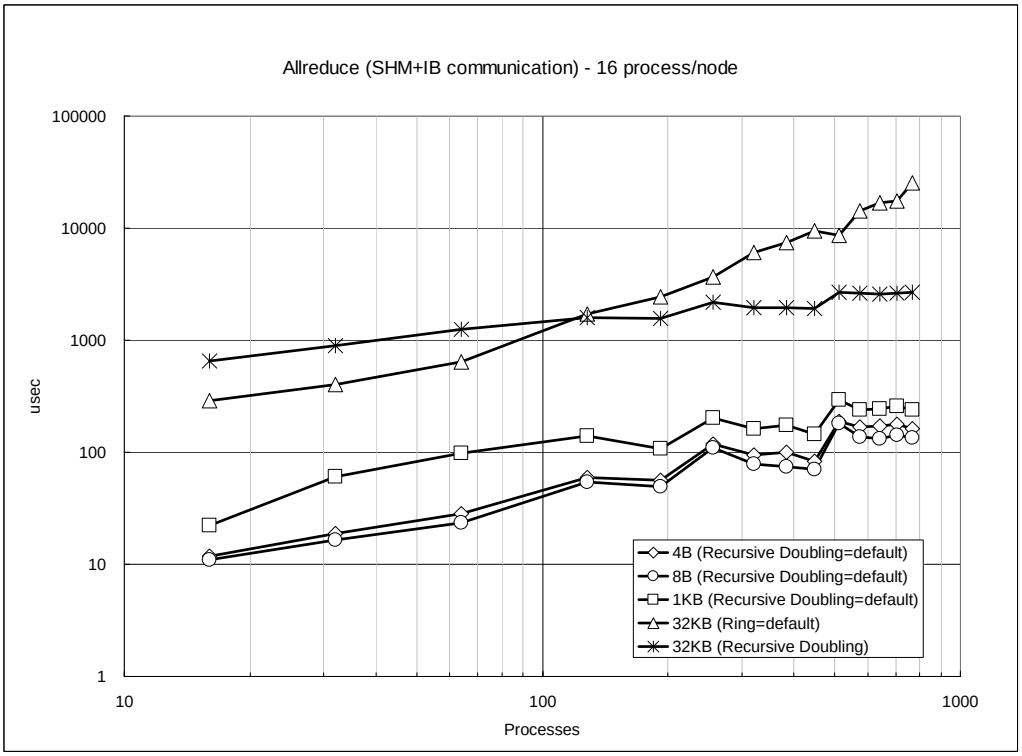
コード	Intel® MPI Benchmarks 3.1
MPI	Open MPI 1.3.3

評価は以下の方法で行った。

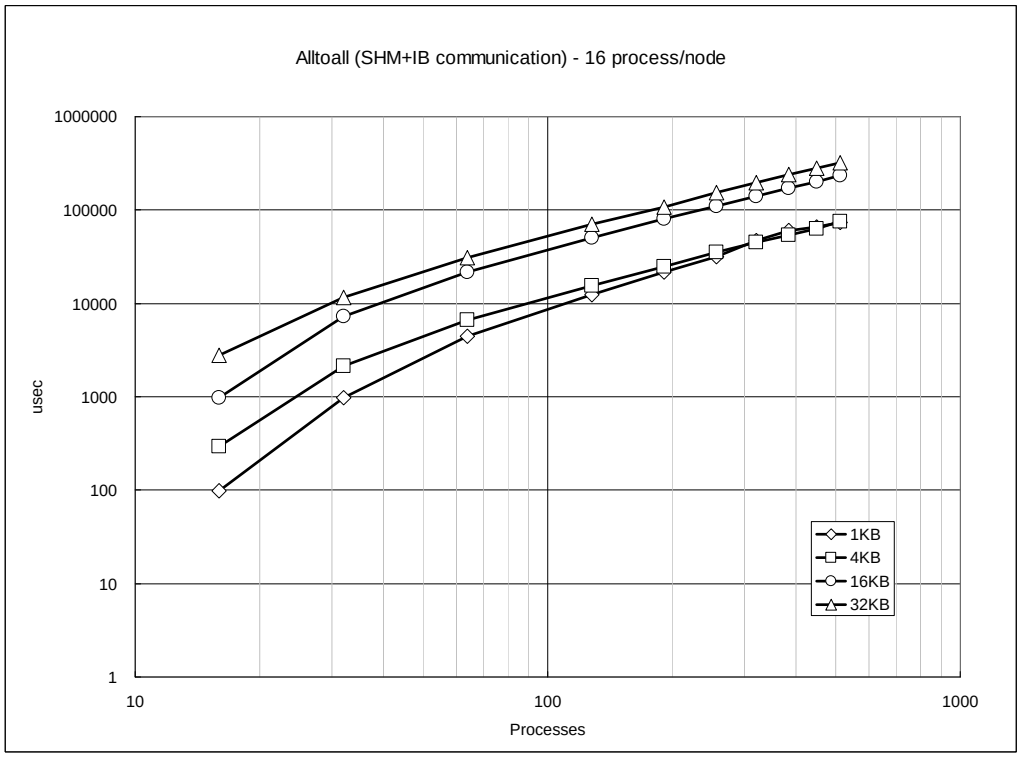
- ノード内のコアを全て使用（1 ノード 16 プロセス× 1 ノード）
- ノード内は共有メモリ通信，ノード間 InfiniBand 通信
- 現状のユーザーアプリケーションを想定し，測定関数と測定範囲を以下のように設定した
  - Allreduce (16～768 プロセス)：4B, 8B, 1KB, 32KB
  - Alltoall (16～512 プロセス)：1KB, 4KB, 16KB, 32KB

検証結果を以下に示す。

- Allreduce (16 プロセス/ 1 ノード)



- Alltoall (16 プロセス/ 1 ノード)



まとめ

- Allreduce

  – 128 プロセス・256 プロセス・512 プロセスの性能値が悪くなるが，相対的にプロセス数とともに実行時間が増えていく。

  ⇒ 一般的に使用される 4 バイトや 8 バイトの Allreduce は，並列度を上げていくと実行コストが見えてくる。このデータ長は一般的にチェックサムに使用されるため，プロセス数増加しても通信量を減らすことができない。このため，スケーラビリティ低下の要因の一つとなる。

  – 32KB では Ring アルゴリズムが選択されるが，実際は 128 プロセスまでは，Recursive Doubling のアルゴリズムの方が効果的である。

- Alltoall
  – プロセス数にスケールして，確実に実行時間が増えていく。
  ⇒ Alltoall を使用したプログラムのスケーラビリティを確保するには，メッセージ長を短くすることが重要となる。

3. ノード間通信とノード内通信

評価に用いたシステムを以下に示す。

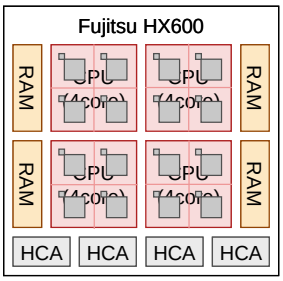
CPU	Quad-Core AMD Opteron™ Processor 8354 2.2GHz 4CPU (16core) / node
RAM	16GB /node
Interconnect	ConnectX DDR HCA * 4
OS	Linux version 2.6.18-92.1.22.el5
Topology	FAT Tree

今回の評価に用いたコードと MPI を以下に示す。

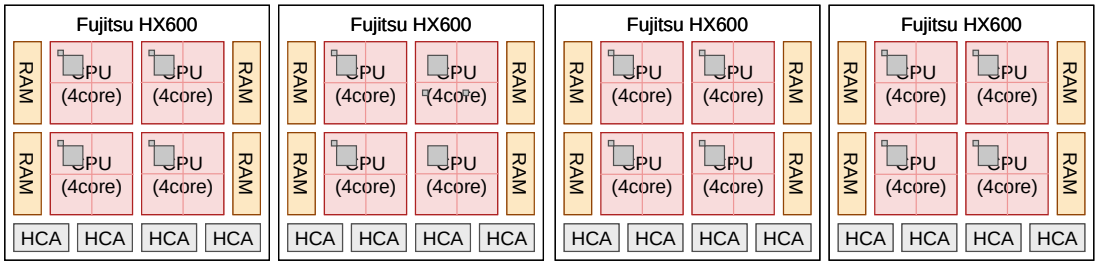
	IB + SHM 通信	IB 通信
実行時 オプション	--mca btl_openib_warn_default_gid_prefix 0 --mca btl self,sm,openib	--mca btl_openib_warn_default_gid_prefix 0 --mca btl self,openib
実行時付加	numactl --interleave=all	numactl --interleave=all

テストパターンと評価システム上のプロセス構成を以下に示す。

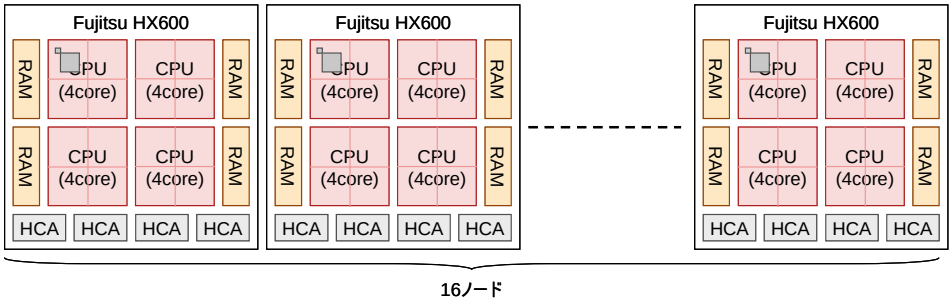
- ノード内のコアを全部使用（1 ノード 16 プロセス× 1 ノード）
  (1) **全ての通信を** SHM **通信**
  (2) **全ての通信を** IB Loopback **通信**



- ノード内の CPU を 1 コアだけ全部使用（4 プロセス/ 1 ノード× 4 ノード）
  (3) **ノード内は** SHM **通信，ノード間は** IB **通信**
  (4) **ノード内は** IB Loopback **通信，ノード間は** IB **通信**

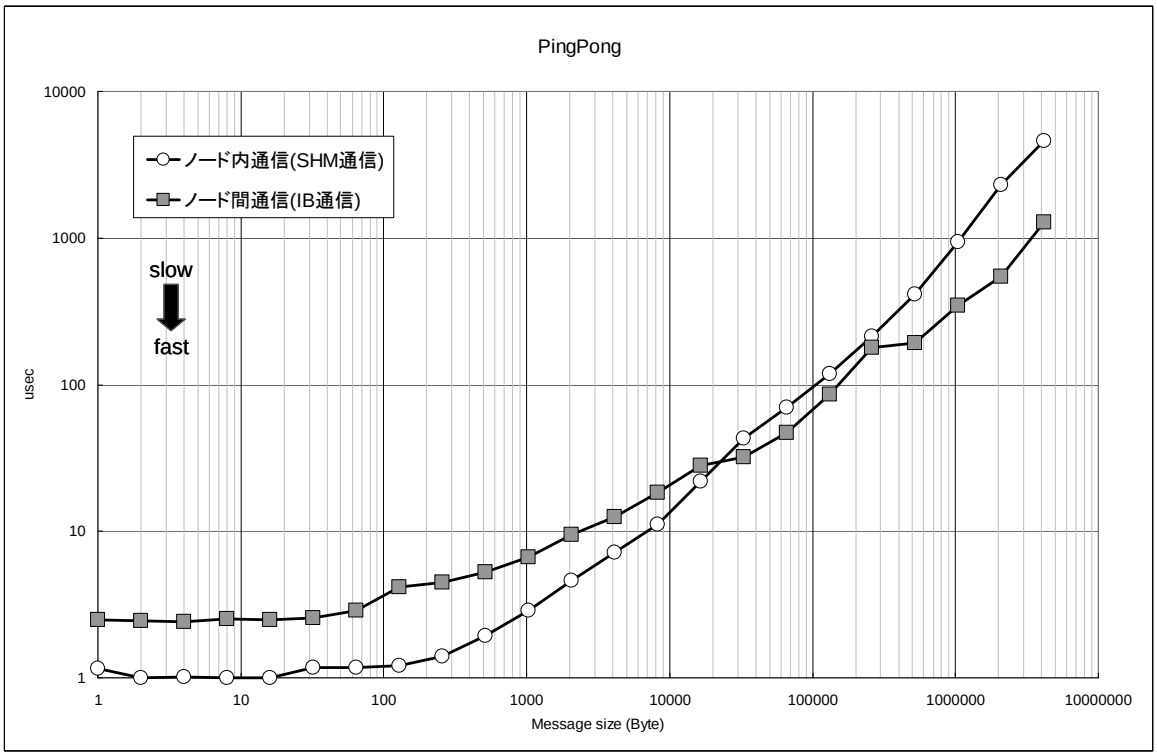


- ノード内の CPU を 1 つだけを使用（1 プロセス/ 1 ノード× 16 ノード）
  (5) **全ての通信を** IB **通信**

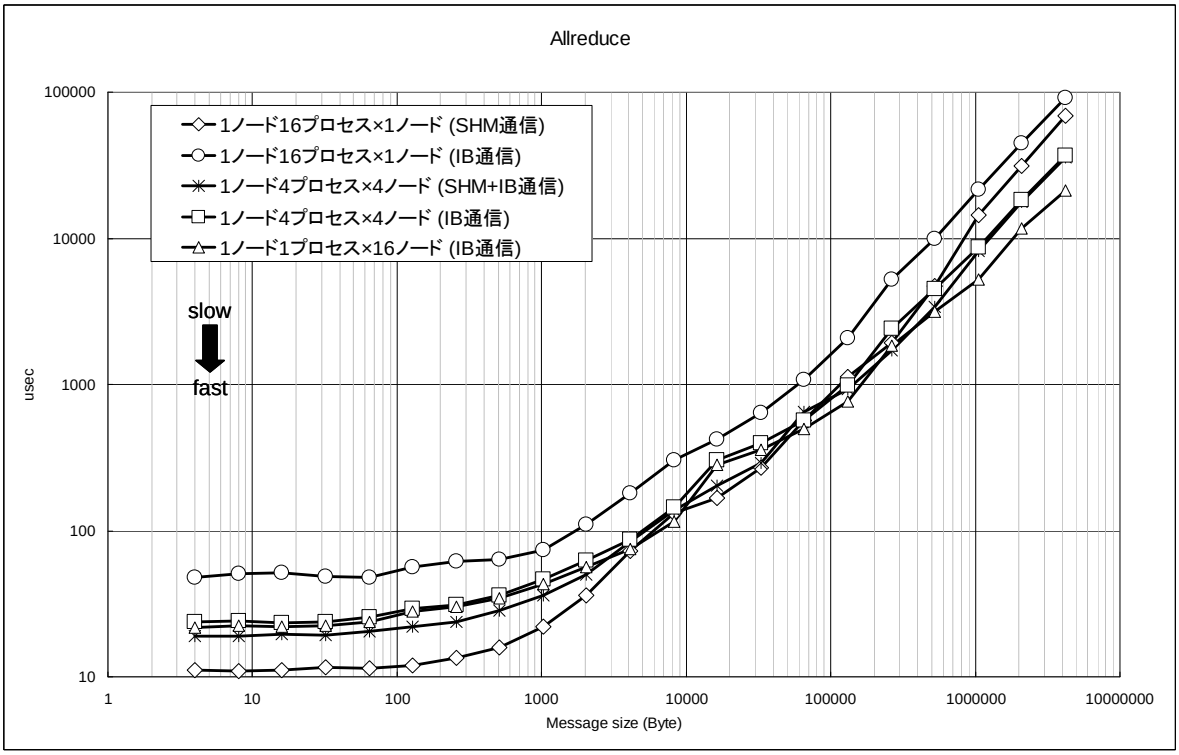


検証結果を以下に示す。

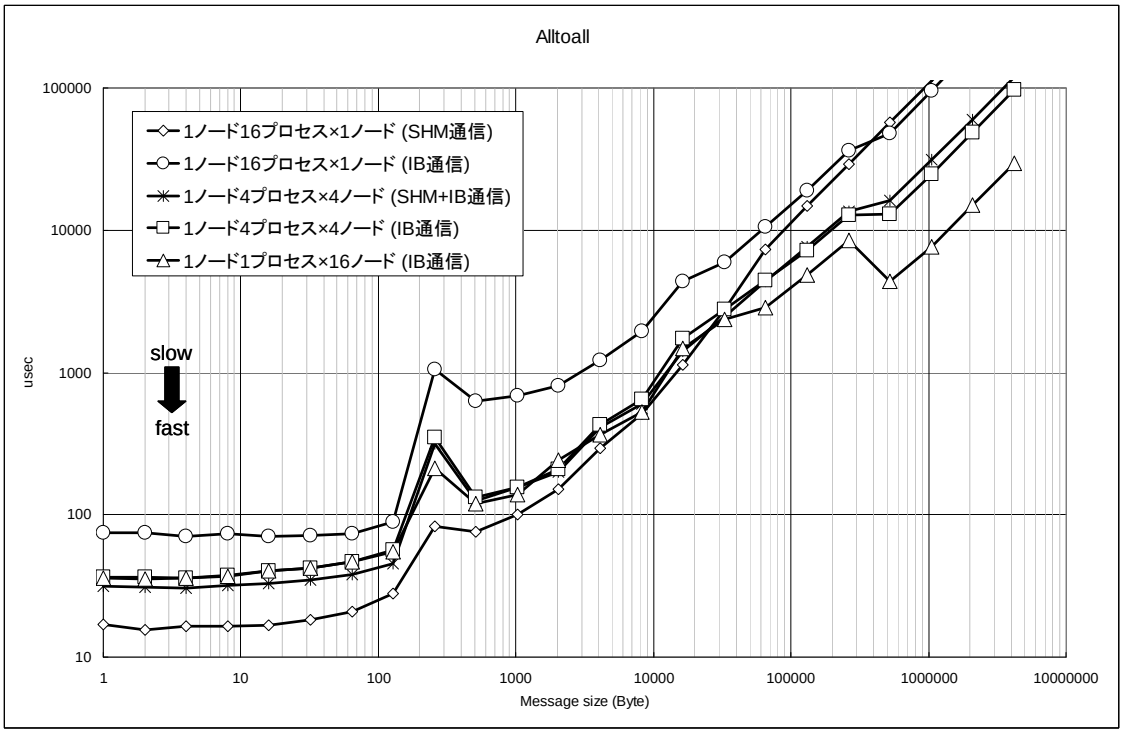
- ノード内通信とノード間通信の比較（PingPong）



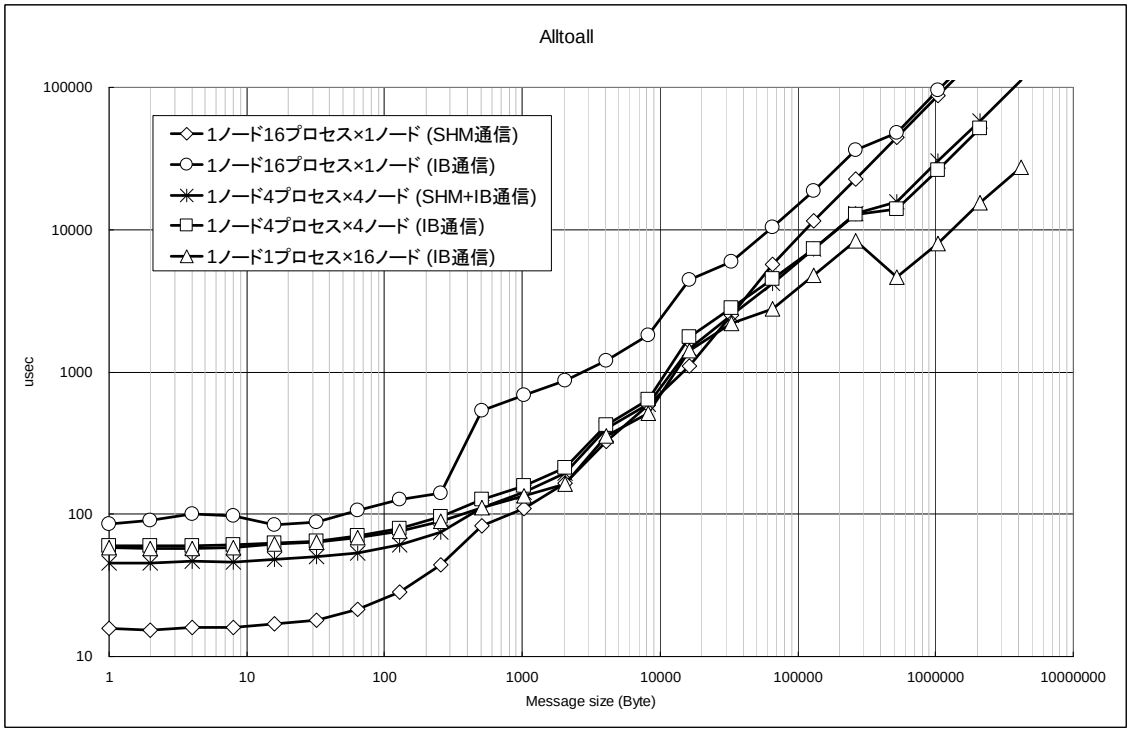
- ノード内通信とノード間通信の比較（Allreduce）



- ノード内通信とノード間通信の比較（Alltoall）



- ノード内通信とノード間通信の比較（Alltoall<一部アルゴリズム変更版>）



まとめ

- 転送長が短い通信では共有メモリ通信が有利となる。本システム(富士通 HX600)では 20KB がしきい値となり, 20KB 以下では共有メモリ通信, 20KB 以上がノード間通信の方が速くなる。
- 必要なメモリやメモリバンド幅が気にならないアプリケーションの場合, アプリケーションの選択は以下のように考える。

	メッセージ長	
	全体的に短い	全体的に長い
ノード内プロセス	ノード内に多くのプロセスを詰め込む	ノード内のプロセスを減らす
ノード内通信	SHM 通信を選択する	IB Loopback 通信を選択する

4. MPI-IO 性能

評価に用いたシステムを以下に示す。

- 計算ノード（富士通 FX1）

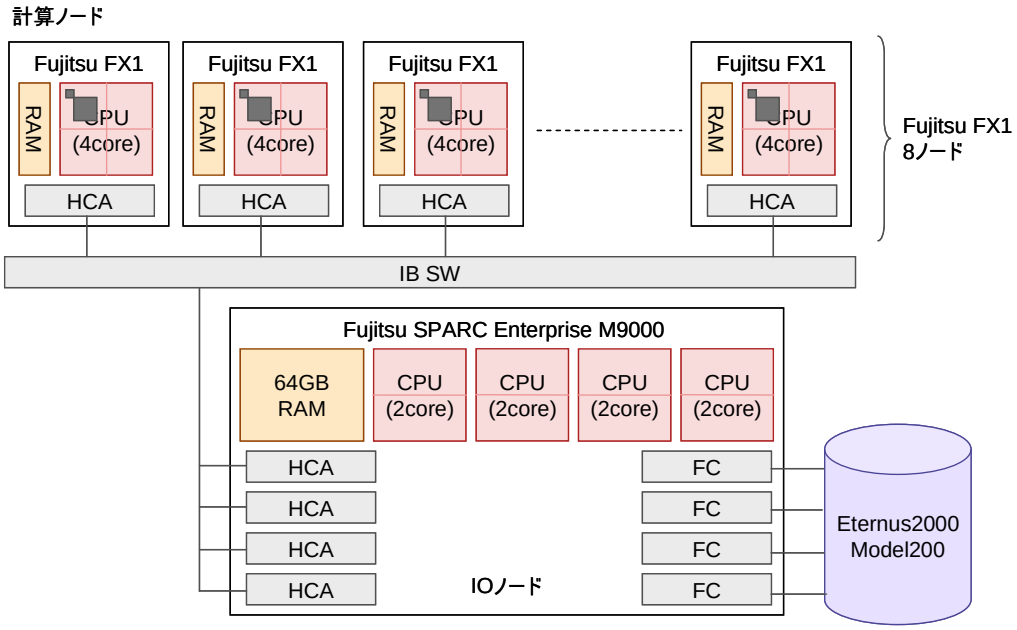
CPU	SPARC64 VII 2.5GHz * 1 / node (4-cores)
RAM	32GB /node
Interconnect	InfiniBand DDR * 1
OS	OpenSolaris Build 79
Middleware	Parallelnavi Base Package 3.1
File system	FUJITSU Parallelnavi SRFS 3.0.0-03(B30000-02G)

- IO ノード（富士通 SPARC Enterprise M9000）

CPU	SPARC64 VI * 4
RAM	64GB /node
Interconnect	InfiniBand DDR * 4
Fibre Channel	4G-FC * 4
File system	Sun StorageTek™ QFS

- ディスクシステム（富士通 ETERNUS2000 Model200）

コントローラ数	2
キャッシュ容量	4GB
Interconnect	InfiniBand DDR * 4
RAID	RAID6(NL-SAS 4D+2P) *
実 WRITE 性能	約 400MB/s



富士通の Parallelnavi Language Package V3 に含まれる MPI を使用した。1 ノードあたり 1 プロセスを生成し，以下の観点でテストを行った。

- 連続データ転送性能
- ストライド転送性能

計測パターンは 2 種類を用意し，以下のように定義する。

- IO キャッシュ性能： IO サーバーへデータを転送した時間とする。
- ディスク込み IO 性能： 実際にディスクに書き込まれるまでの時間とする。

以下に，連続データの転送性能について評価する。
連続データの転送性能は，以下の API 間の消費時間を計測した。

呼び出すAPI一覧

MPI-IO	Split Files	IOマスター	
MPI_File_open	open	open	
MPI_File_set_view		MPI_Gather	IOキャッシュ性能
MPI_File_write	write	write	
MPI_File_sync	fsync	fsync	ディスク込みIO性能
MPI_File_close	close	close	

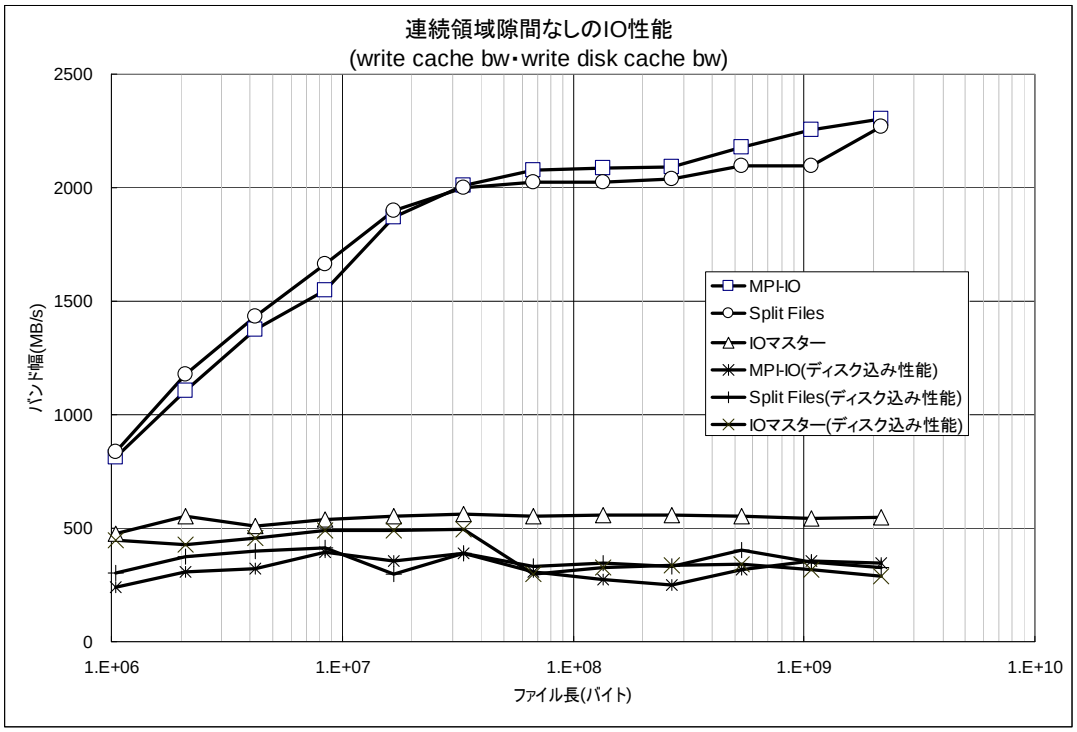
		I/O 方式		
		MPI-IO	Split Files	IO マスター
ディスク込み IO 性能	IO キャッシュ性能	MPI_File_set_view MPI_File_write	write	MPI_Gather write (rank0 のみ)
		MPI_File_sync	fsync	fsync (rank0 のみ)

IO 性能は性能ブレが激しく，数回の試行に対する平均値を性能値と定義することが難しい。例えば，IO キャッシュの吐き出しタイミングによって大きく性能ブレが発生するが，ユーザープログラムから見るとランダムに発生するため想定できない。
計測は，1 つの IO 長に対し 20 回試行し，明らかに性能ブレしたデータを目視で排除して集計した。今回提示するデータは，各方式を比較するために「システムが安定しているときの理想的な IO 性能」と考える。

連続データ転送性能は，以下の 3 つのパターンについて計測を行った。

Split Files 方式	各プロセスがファイルを作成する。
MPI_IO 方式	MPI-IO のブロッキング入出力を用いて，一つのファイルを作成する。
IO マスター方式	MPI_Gather によって，全プロセスのデータをルートプロセスに集めてから，一つのファイルを作成する。

検証結果を以下に示す。



MPI-IO 性能と Split Files 性能はほとんど変わらずに，ファイル長が長くなるにつれて傾きは悪くなるがスケールしていく。IO マスター方式は，500MB/s 程度で一定となっている。これは，データの再構成に時間がかかるため，性能面で劣ると言える。

今回評価したシステムでは，ディスク性能が低いため，ディスク込み性能は全ての方式でほぼ同じ性能になっている。MPI-IO 方式・Split Files 方式ともに，IO キャッシュ効果を有効に利用していると言える。

以下に，ストライドデータの転送性能について評価する。

ストライドデータ転送性能は，以下の 3 つのパターンについて計測を行った。

MPI-IO 方式	MPI-IO のブロッキング入出力を用いて，一つのファイルを作成する。
MPI_IO(COLL)方式	集団的 MPI-IO を用いて，一つのファイルを作成する。
IO マスター方式	MPI_Gather によって，全プロセスのデータをルートプロセスに集めてから，一つのファイルを作成する。

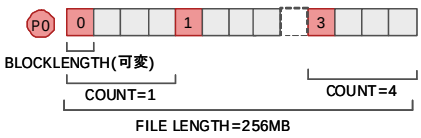
計測は，IO キャッシュ性能を測定し，以下の API 間の消費時間を計測した。

呼び出すAPI一覧

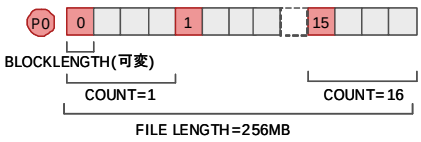
MPI-IO	MPI-IO(COLL)	IOマスター	
MPI_File_open	MPI_File_open	open	
MPI_File_set_view	MPI_File_set_view	MPI_Gather	IOキャッシュ性能
MPI_File_write	MPI_File_write_all	write	
MPI_File_sync	MPI_File_sync	fsync	
MPI_File_close	MPI_File_close	close	

【ストライド転送性能検証パターン】

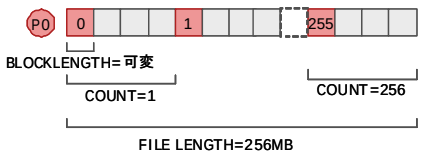
1. ファイル長256MB、ブロック粒度(ファイル長/繰り返し数/プロセス数)、繰り返し数 (4回)



2. ファイル長256MB、ブロック粒度(ファイル長/繰り返し数/プロセス数)、繰り返し数(16回)

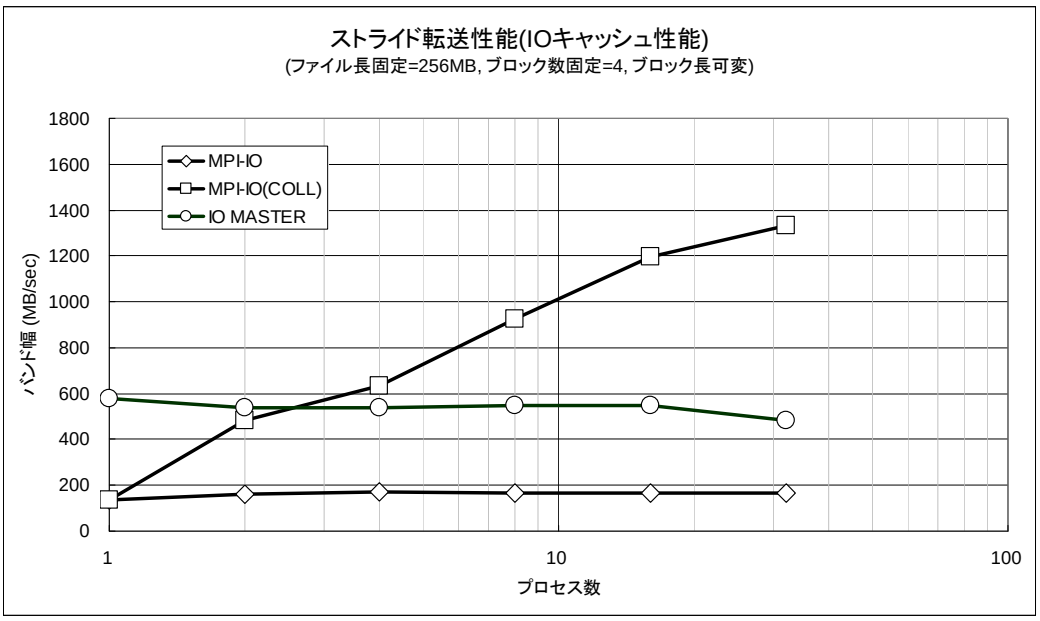


3. ファイル長256MB、ブロック粒度(ファイル長/繰り返し数/プロセス数)、繰り返し数(256回)

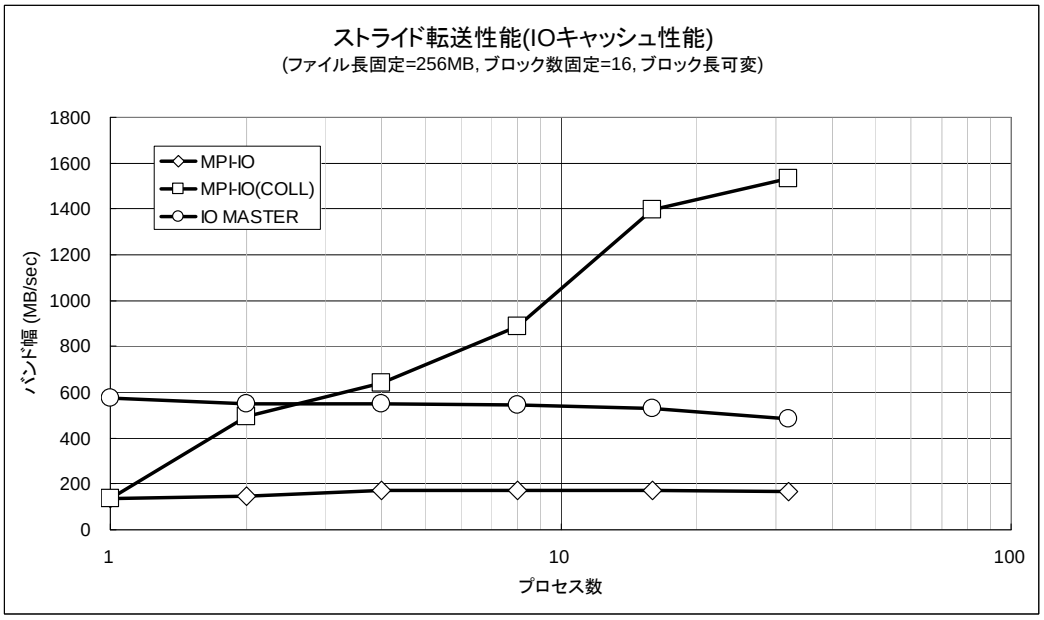


検証結果を以下に示す。

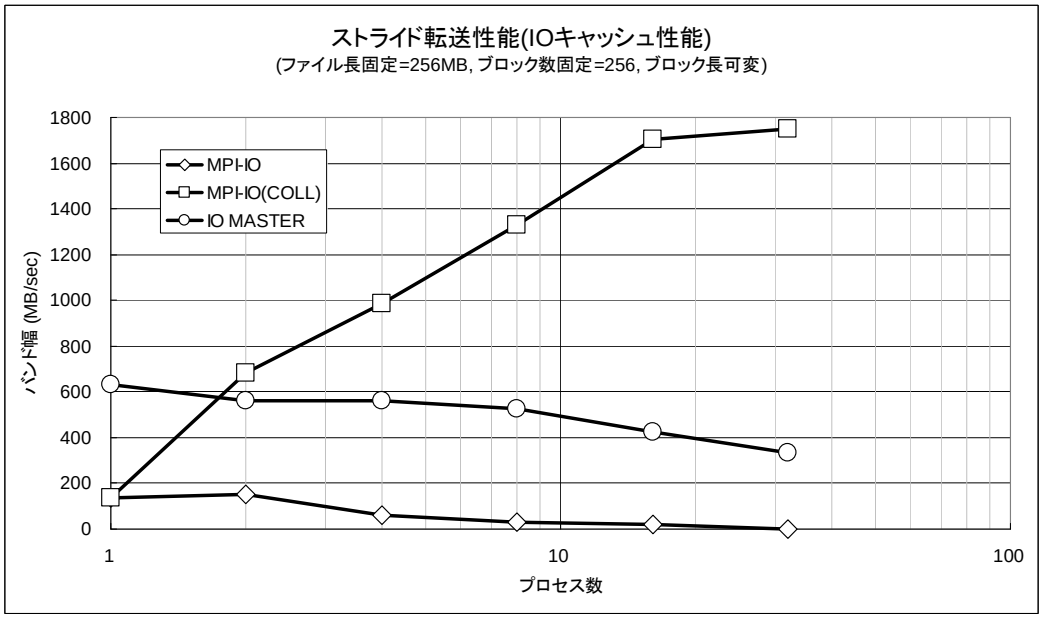
＜ストライド転送性能(ブロック数=4)＞



＜ストライド転送性能(ブロック数=16)＞

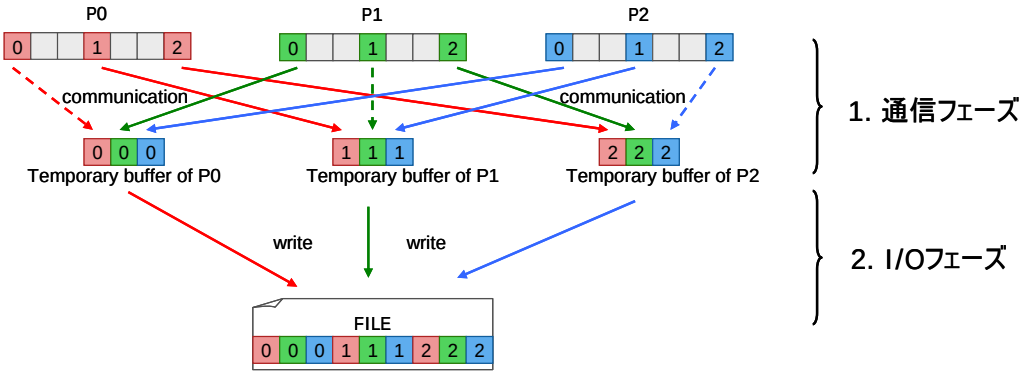


＜ストライド転送性能(ブロック数=256)＞



ストライドデータのファイル転送では，集団的 MPI-IO の性能が効果的である。16 プロセス実行の比較的小さな環境ではあるが，転送性能はスケールしている。これは，ROMIO における集団的 MPI-IO の実装に，Data Sieving の技術が利用されていることが効果的に働いていると考えられる。Data Sieving とは，ストライドデータの入出力処理において，各プロセス内にバッファリングを行い，連続データでファイルを入出力する方法である。2 つのフェーズにわかれていて，出力の場合は通信フェーズから I/O フェーズ，入力の場合は IO フェーズから出力フェーズの順に処理を行う。Data Sieving を利用することで，一つ一つの小さなブロックに対するファイル要求ではなく，大きなブロックに対するファイル要求が行われるため，ブロック数が多い場合に効果的な性能が期待できる。

Data Sieving のファイル出力処理を以下に示す。



逆に，非集団的 MPI-IO の性能は，ブロック数を増やすにつれて性能の低下が大きい。各プロセスが独立して IO を実行する上で，細かい領域に対してのファイルアクセスに対する排他処理の影響が大きいと考えられる。

以上

2.2.3. 書式付 I/O 性能

上智大学　南部伸孝
富士通株式会社　内藤俊也、杉崎由典

1. はじめに

I/O 処理に要する時間の内訳は、システムの実 I/O 時間＋ランタイムの書式処理時間となっている。

real*8 write プログラムを用いた書式付 I/O 時間を富士通コンパイラで評価した所、システム時間が約 12%であり、ランタイム時間が約 88%ということが確認された。

即ち、書式付 I/O 性能は、ランタイムの性能に大きく影響される。ここでは、ランタイムの書式付 I/O 処理性能という観点で評価を行う。

2. 測定プログラム

以下のプログラムを用いた。

【real*8 プログラム】
r8(7500000) ⇒５７．２Ｍバイト

●real*8 write

```
open(11,file='iof_r8.dat',form='formatted',status='new',err=999)
call gettod(t(1,1))
write(11,'(4d24.15)') r8
call flush(11)
call gettod(t(2,1))
close(11)
```

●real*8 read

```
open(11,file='iof_r8.dat',form='formatted',status='old')
call gettod(t(1,2))
read(11,'(4d24.15)') r8
call flush(11)
call gettod(t(2,2))
close(11)
```

上記は real*8 の例であるが、型別にそれぞれ write と read を測定した。

また、各型毎の I/O 対象のファイルのサイズを以下に挙げる。ただし、ファイルのサイズは、配列サイズではなく、実際に入出力処理する対象のファイルのサイズを示している。

表 1.　入出力対象のファイルサイズ

	INTEGER*4	REAL*4	REAL*8	COMPLEX*8	COMPLEX*16
要素数 7.5M 個の I/O 対象 Filesize(byte)	75750000	181875000	181875000	363750000	363750000

3. 評価環境

評価したマシン環境及びコンパイラを以下に挙げる。

表 2.　評価対象コンパイラ

コンパイラ	バージョン
富士通	3.2／3.0
Intel	10.1
PGI	7.1－3
PathScale	3.3.1
日立	－

表 3.　評価対象マシン環境

	PRIMERGY RX200 S3	SR11000	SR11000	SR16000	FX1	HX600	PRIMEQUEST 580
ﾌﾟﾛｾｯｻ	Woodcrest	POWER5	POWER5+	POWER6	SPARC 64 VII	Optero n	Itanium2
周波数	3.0GH z	1.9GHz	2.3GH z	4.7GHz	2.52GHz	2.3GHz	1.6GHz
ﾌｧｲﾙｼｽﾃﾑ	ext3/srfs	mmfs	mmfs	nfs3,mmfs	srfs/nfs/ tmpfs	nfs/ext2	srfs/tmpfs

4. 測定結果及び評価

評価の観点として以下の点が挙げられる。
- ・変数の型
- ・total write/read 性能
- ・マシン
- ・ファイルシステム

それぞれの観点で評価を行った。

4.1 型別の性能評価

評価する I/O の write/read 種別と変数型として以下を対象とした。
- ・integer*4　write/read
- ・real*4　write/read
- ・real*8　write/read
- ・complex*8　write/read
- ・complex*16　write/read

各社のコンパイラを用いて、PRIMERGY RX200S3 上で測定を行った。
結果は以下の通り。

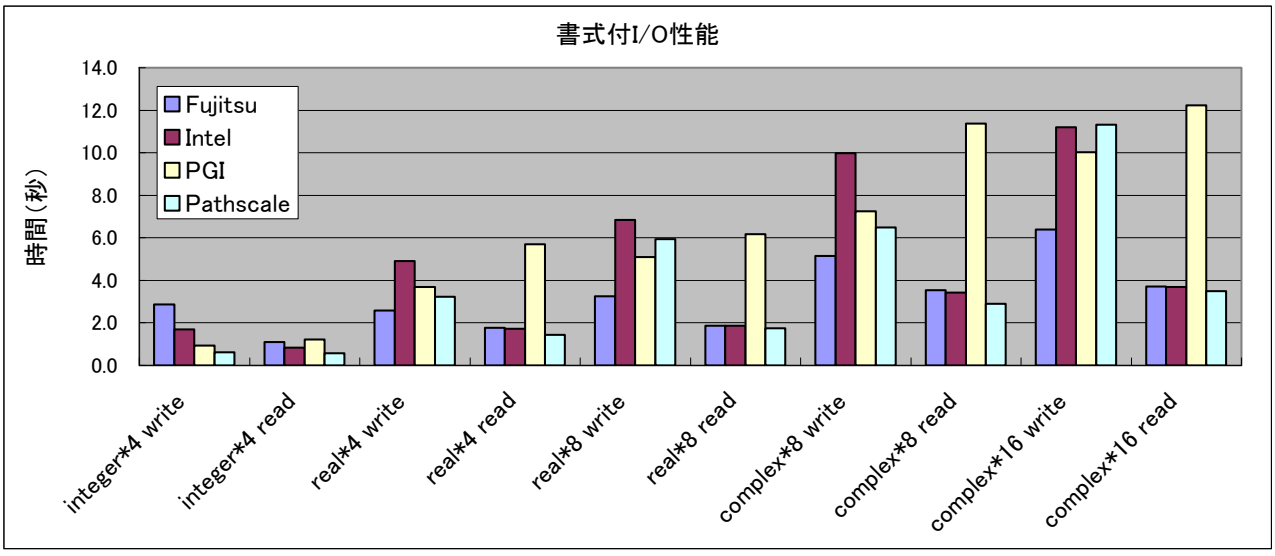


図 1. 各社のコンパイラで型別の測定結果

富士通コンパイラは、real,complex 型で良い結果となっている。

4.2 Total,write/read 性能評価

前節で測定した値に対して、以下のように結果時間を合計して評価した。
- ・各型の write 合計時間
- ・各型の read 合計時間
- ・各型の write/read 合計時間

各社のコンパイラを用いて、PRIMERGY RX200S3 上で測定を行った。
結果は以下の通り。

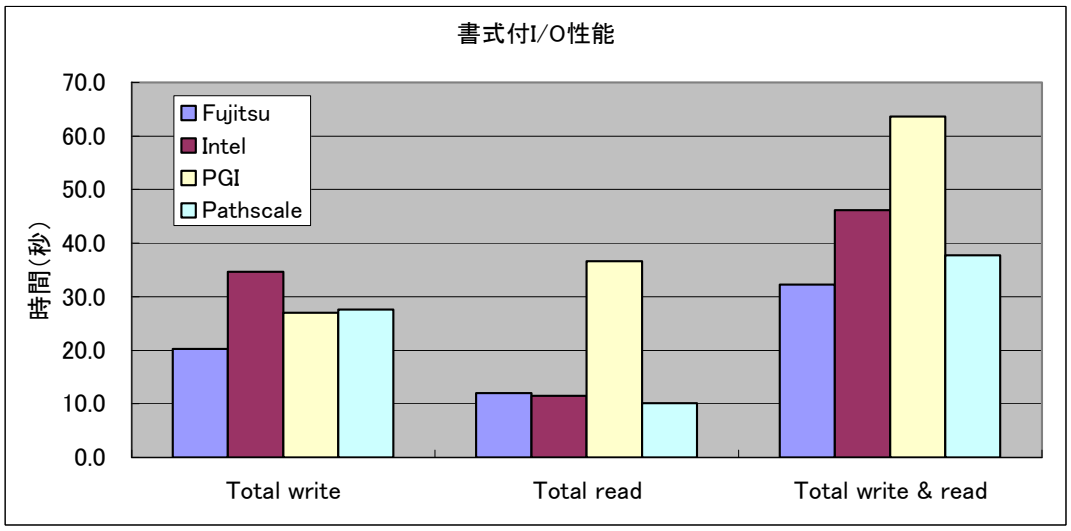


図 2. 各社のコンパイラで write/read 合計の結果

富士通コンパイラは、合計時間でも良い結果となっている。

4.3 マシン毎の型別の性能評価

各社のマシンで変数型それぞれに対して評価を行った。また、複数のファイルシステムで測定した。対象マシン及びファイルシステムは以下の通り。

- PRIMERGY RX200S3 ext2
- PRIMERGY RX200S3 srfs
- SR11000 mmfs
- SR16000 nfs3

各社のコンパイラを用いて、測定を行った。
結果は以下の通り。

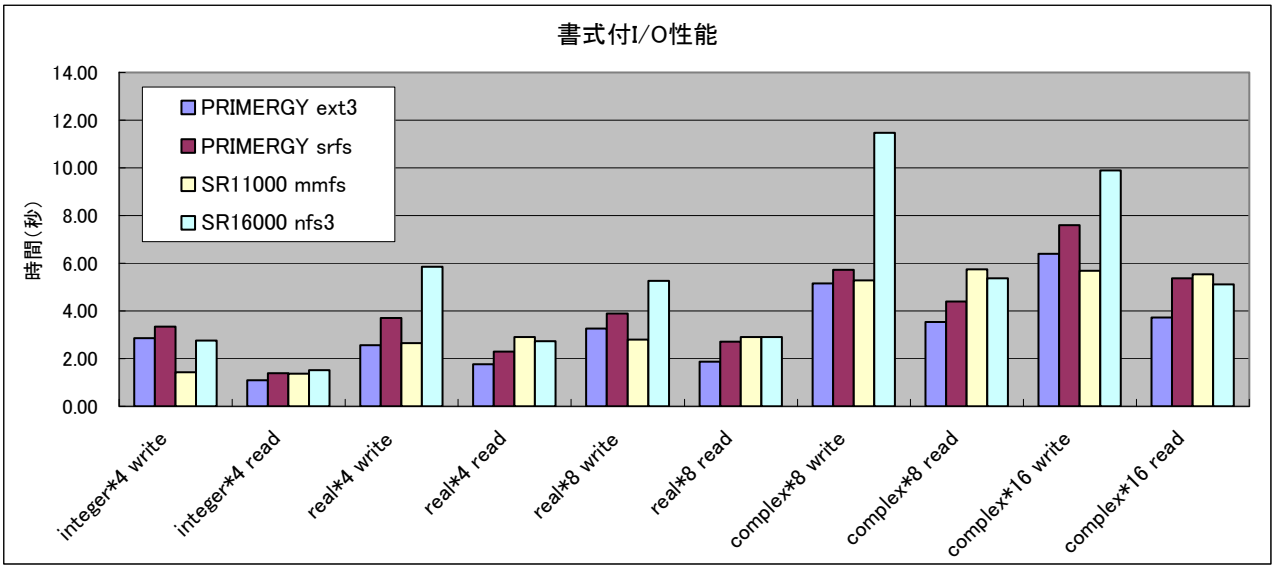


図 3. 各社のマシン及びファイルシステムで型別の測定結果

PRIMERGY は read で良い性能、SR11000,SR16000 は nfs3 よりも mmfs の方が良い性能となっている。

九州大学のシステムである PRIMERGY RX200S3(srfs)と SR11000(mmfs)で比較すると、write 性能では SR11000 が有利、read では PRIMERGY RX200S3 が有利となったが、性能差は大きくなく、ほぼ同等の性能といえる。

4.4 マシン毎の total,write,read 性能評価

前節で測定した値に対して、以下のように結果時間を合計して評価した。

・各型の write 合計時間
・各型の read 合計時間
・各型の write/read 合計時間

各社のコンパイラを用いて測定を行った。
結果は以下の通り。

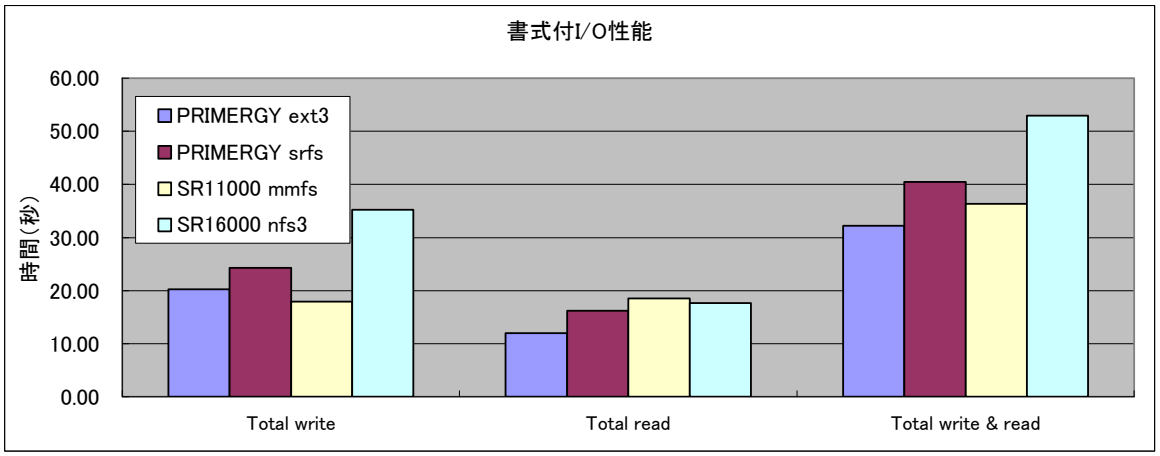


図 4. 各社のマシン及びファイルシステムで write/read 合計の結果

total で見ると、nfs3 以外はほぼ同等の性能である。

複数のマシンを利用して、I/O 性能の測定を行ったが、I/O 性能はファイルシステムやメモリの性能に大きく依存するため、同じシステムで評価する必要がある。

4.5 ファイルシステム毎の型別評価

ファイルシステムを変化させての測定を実施した。
各社のマシン及びファイルシステムにおいて、変数型それぞれに対して評価を行った。
結果は以下の通り。

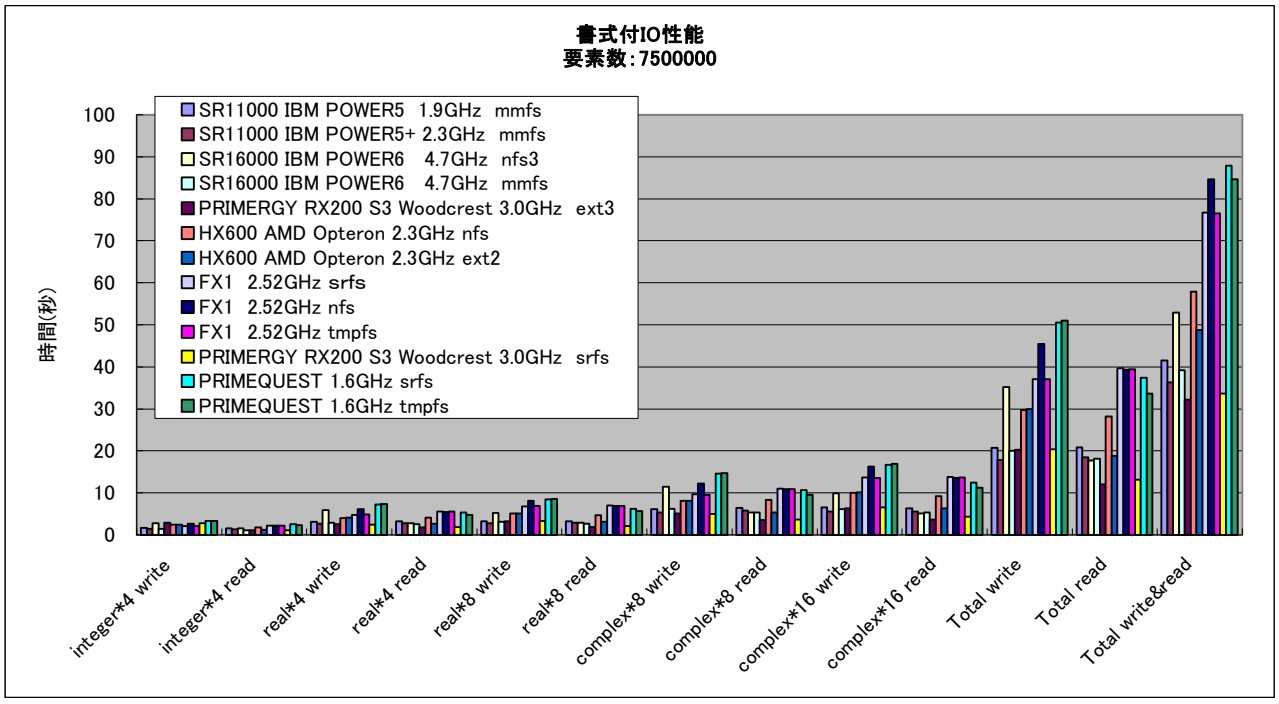


図 5. ファイルシステムで型別の測定結果

同一マシンでは、tmpfs,srfs が良好であり、nfs は遅いという結果となった。

次に、上記の結果から、各社のマシン毎に結果を抽出して、同一マシン上でのファイルシステム毎の性能を評価する。

(1)SR11000,SR16000 抽出

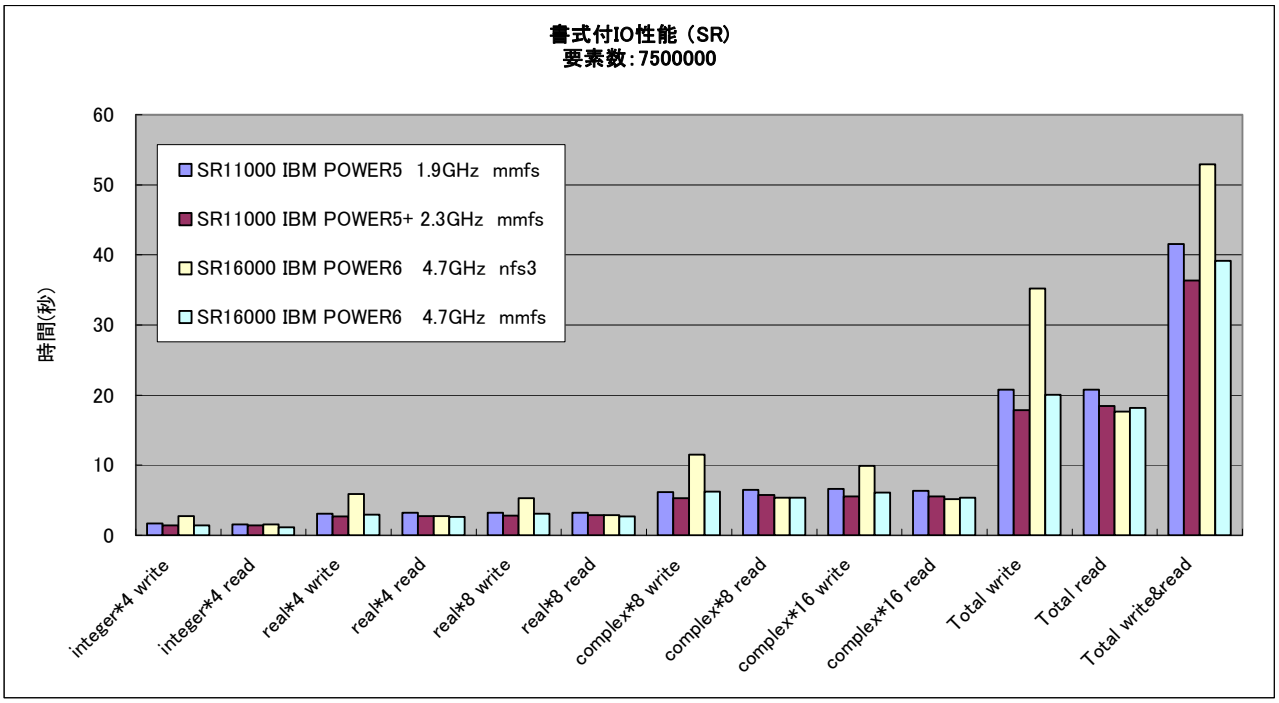


図 6. ファイルシステムで型別の測定結果（SR 抽出）

mmfs と nfs を比較すると、mmfs の方が write 性能では約 1.72 倍速い結果となった。

(2) FX1 抽出

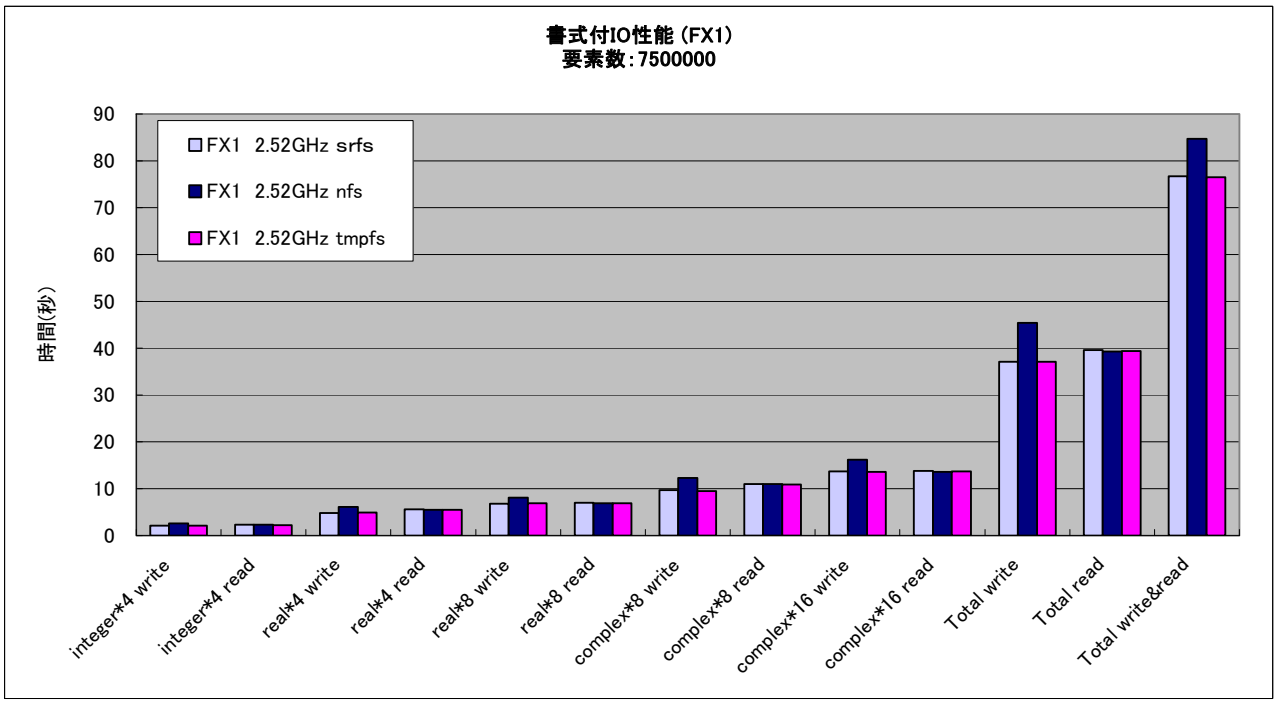


図 7. ファイルシステムで型別の測定結果（FX1 抽出）

srfs と nfs を比較すると、srfs の方が write では約 1.22 倍速い結果となった。

(3) PRIMERGY,HX600,PRIMEQUEST 抽出

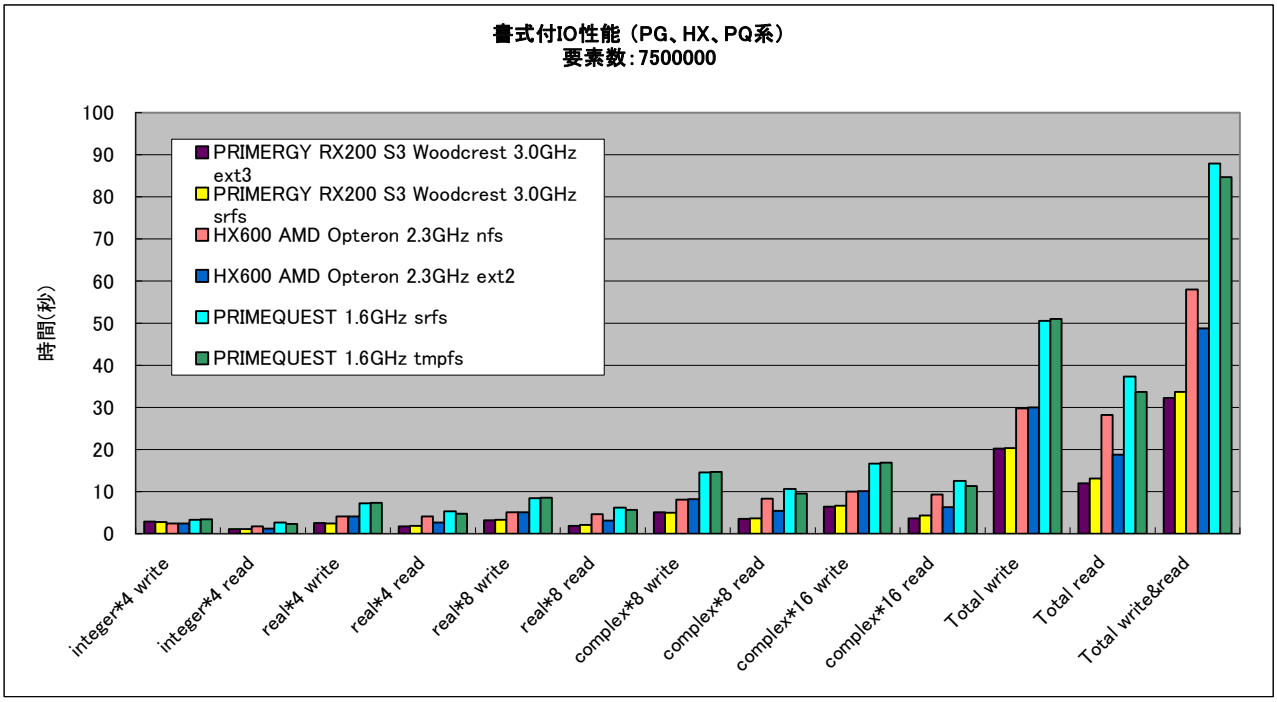


図 8. ファイルシステムで型別の測定結果（PRIMERGY,HX600,PRIMEQUEST 抽出）

4.6 ファイルシステムを統一した際のマシン機種毎の性能評価

これまでの測定結果から、ファイルシステム種別を同一（高速ファイルシステム：srfs,mmfs）にすることにより、ファイルシステムの性能差の影響を排除した性能評価を行った。

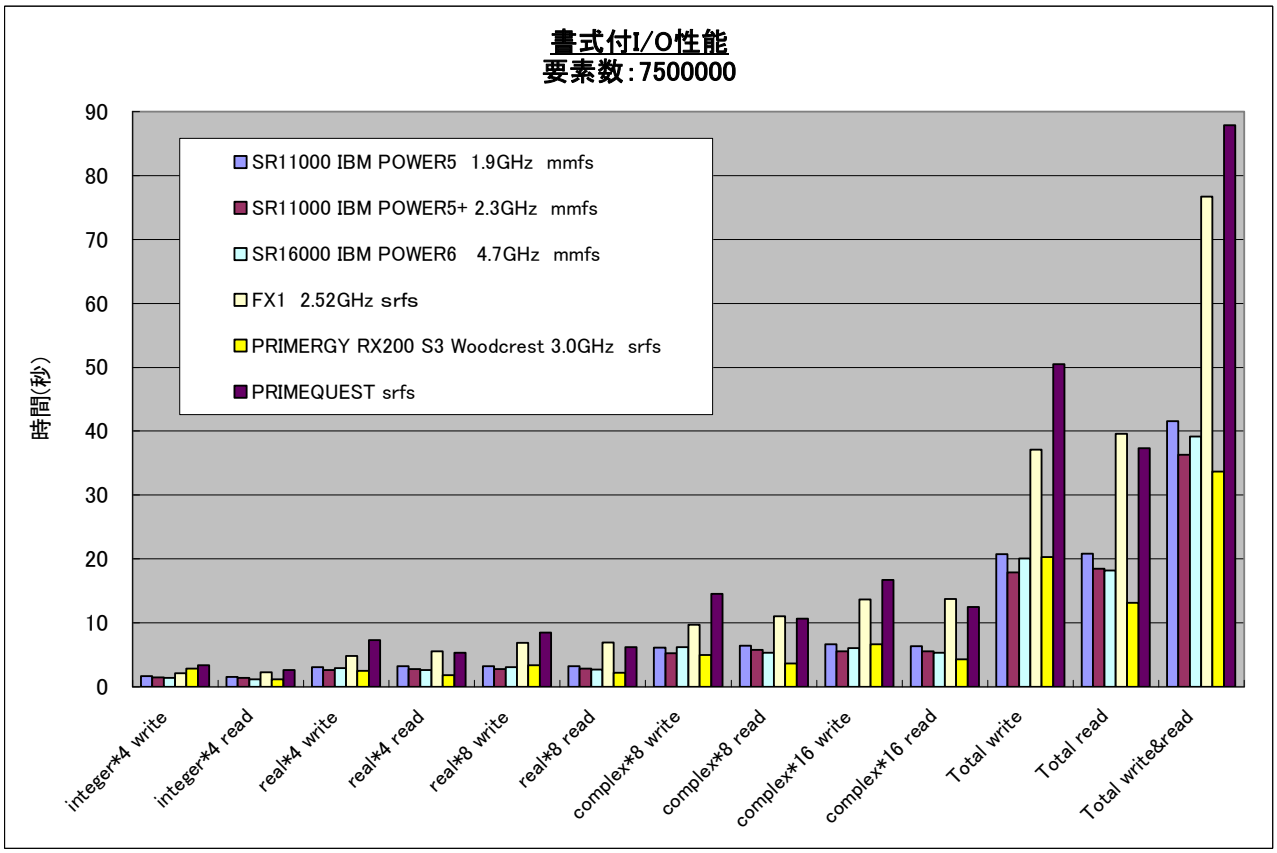


図 9. ファイルシステムを同一としたマシン別の性能比較

PRIMERGY は read で若干良く、SR11000 は write で若干良い。ただし、両者の性能はほぼ同等といえる。

上記の評価では、CPU 周波数の高いマシンが有利となってしまう。書式付 I/O 処理がランタイムの書式処理の性能差であることと、ランタイムの処理は CPU 周波数の影響が大きいことを考慮する必要がある。

そこで、CPU 周波数の低いマシンの性能を補正し、CPU 周波数の違いによる影響を排除した。PRIMERGY RX200S3 の CPU 周波数比を乗算することで補正し、PRIMERGY RX200S3 の性能を 1 とした相対的な性能を比較した。

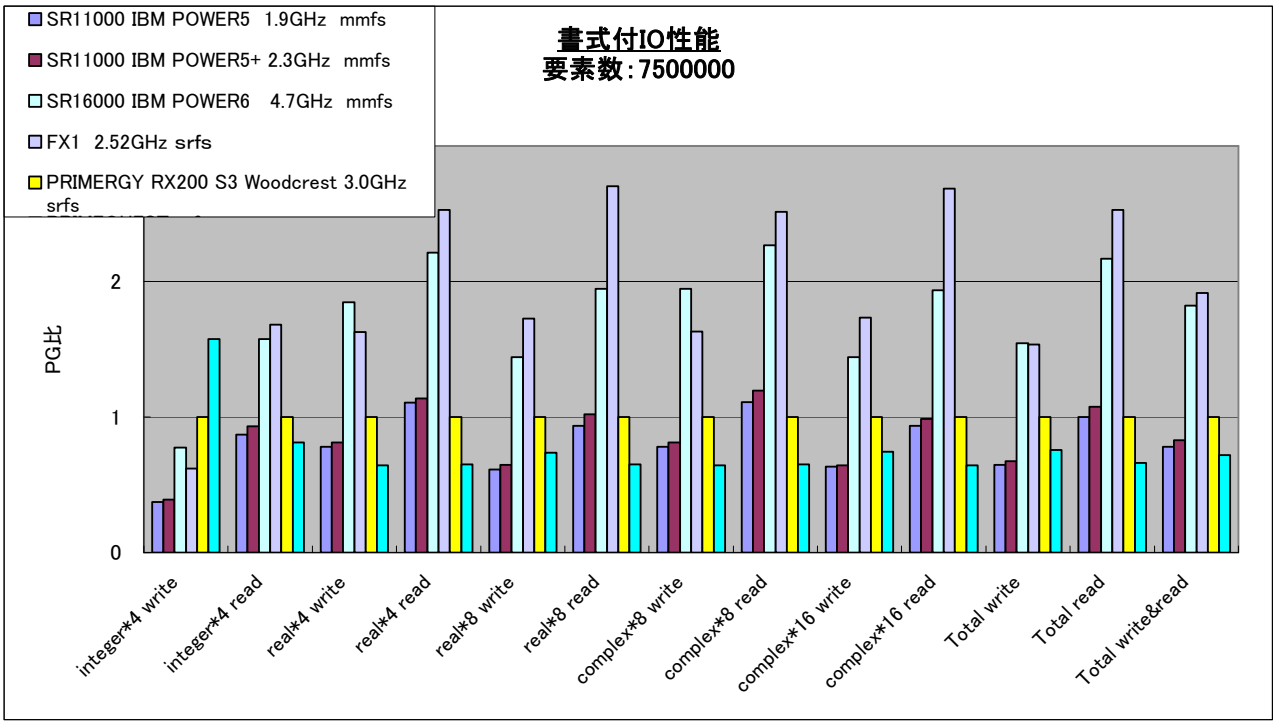


図 10. ファイルシステムを同一としたマシン別の性能比較（周波数比補正）

補正した結果から、PRIMERGY は周波数比で見ると妥当な性能といえる。（ただし、integer*4 は除く）

SR16000 は CPU 周波数の割には性能が出ていない。また、FX1 は浮動小数点の処理が遅いという結果となった。

4.7 測定範囲について

書式付 I/O 処理の測定が、システムの I/O 処理及びランタイム処理のどの範囲まで含まれるのかについて述べる。

(1)書式付 write 性能

RTS 性能（書式変換処理＋バッファへのコピー）＋I/O キャッシュ性能

注）fflush は I/O キャッシュへの flush であり、ディスクへの書込でない。（富士通のシステムではこの様な処理となっている）

(2)書式付 read 性能

ディスクからの読込処理＋RTS 性能（書式変換処理）

注）open 文の処理は、最初の read 文で実行される。そのため、read には open 文のコストも含まれてしまう。（富士通のシステムではこの様な処理となっている）

表 4. 書式付 I/O 処理のコスト内訳

		RTS 処理	RTS(%)	Fflush	Fflush(%)
FX1	Write(秒)	2.044661	99.94 %	0.001131	0.06 %
	Read (秒)	2.285298	100.00 %	0.000001	0.00 %
SR11000 (POWER5)	Write(秒)	1.789408	99.58 %	0.007491	0.42 %
	Read (秒)	1.566522	100.00 %	0.000001	0.00 %

4.8　Sun コンパイラとの比較

FX1 の性能が遅い点に関して、コンパイラの違いによる性能差を確認するため、SUN コンパイラと富士通コンパイラの結果と比較を行った。また、富士通コンパイラの次版は、FX1 のハード機能(prefetch)を意識した性能向上を行ったライブラリであり、こちらも合わせて比較した。

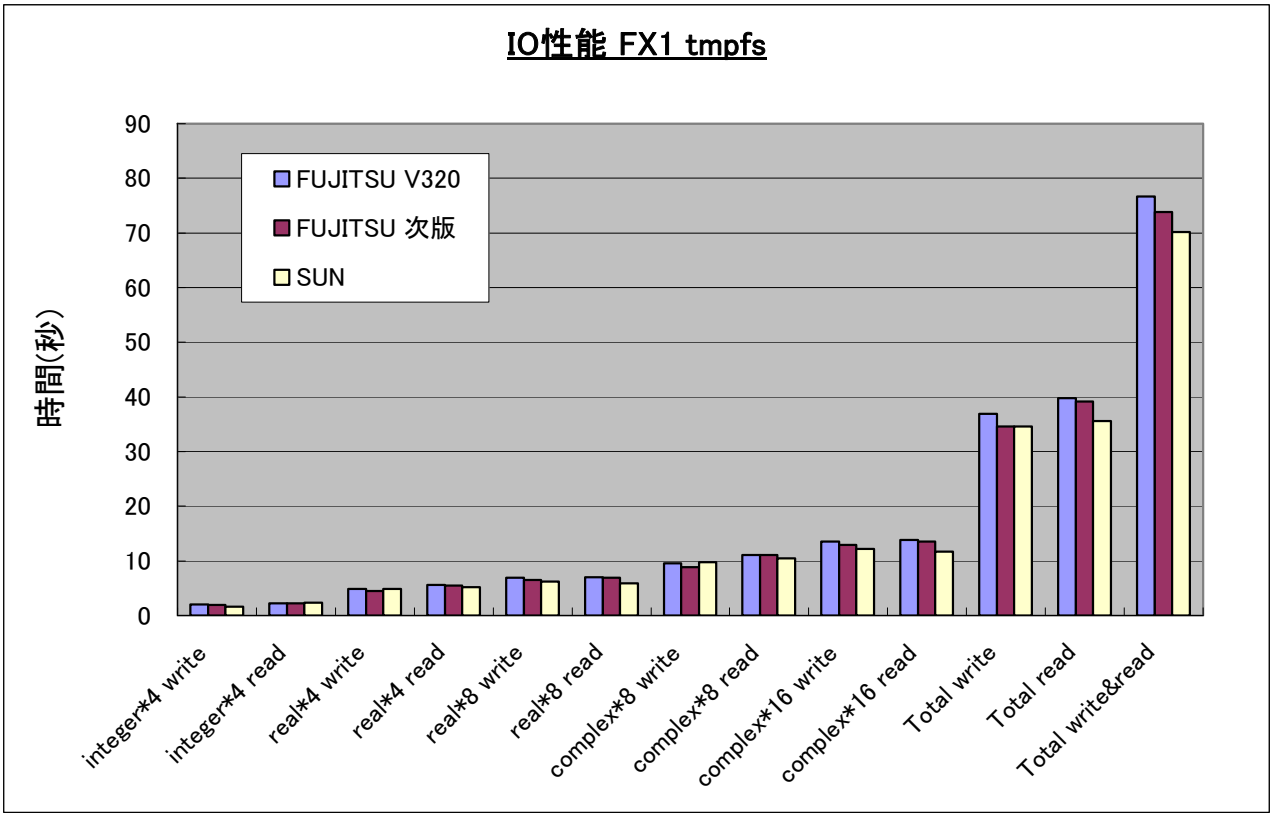


図 11. SUN コンパイラとの性能比較

結果から、多少の性能の違いはあったが、大きな性能差は確認出来なかった。

4.9　FX1 と PRIMERGY との性能差

まず、書式付入出力文で動作するランタイムの機能とコストの割合を調査した。結果を以下に示す。

書式付入出力文で動作する機能とその割合：

- 書式付入出力部分制御、文の組み合わせチェック、データ転送　(5%)
- 書式解析とチェック　(10%)
- データ編集　(15%)
- データ変換　(70%)

データ変換処理のコストが高く、この処理の差が大きいと推測する。

次に、FX1 と PRIMERGY の性能差を real*8 write と real*8 read でそれぞれ比較した。

real*8 の write 部分で約 2 倍、read 部分で約 3 倍の性能差がある。(クロック比を加味した値では、write 部分で約 1.6 倍、read 部分で約 2.65 倍の性能差がある)

ランタイムシステムライブラリの中で性能差が顕著な部位について調査すべきである。

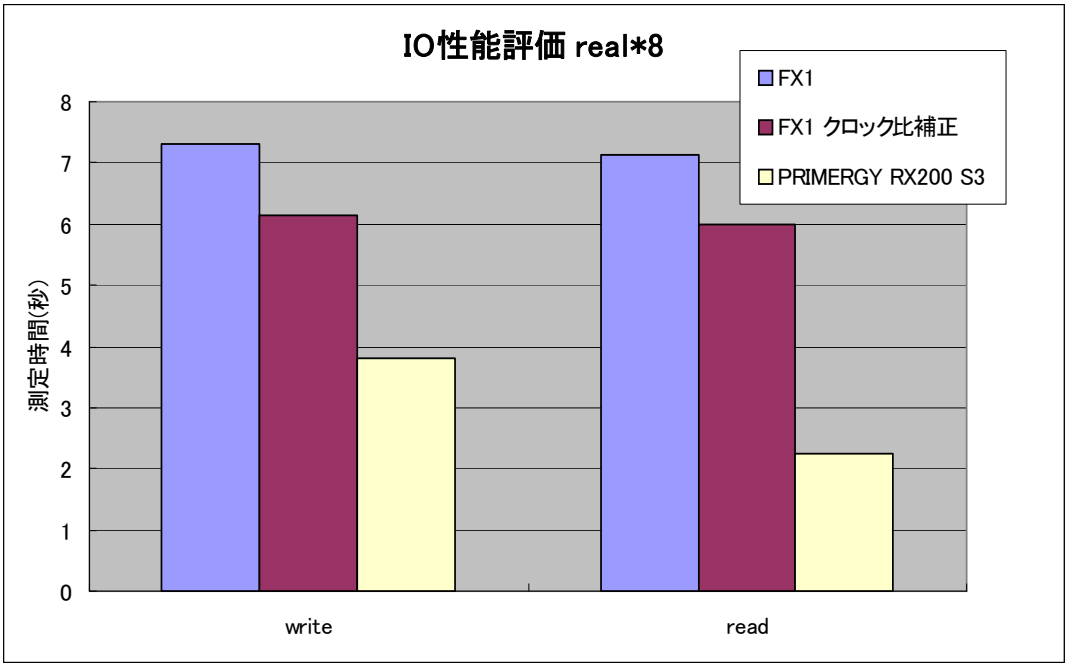


図 12. real*8 型 write/read 性能の FX1 と PRIMERGY との性能比較

(1) real*8 write

real*8 write 処理の中で、ライブラリコストが大きいのは、E 型変換処理手続き。E 型変換処理手続き以外の部分では、性能差はない。

表 5. write ライブラリコスト比較 1

	FX1(クロック比補正)	PRIMERGY RX200S3	性能差
E 型変換処理	4.84 秒	2.27 秒	1.91 倍
これ以外の処理	1.31 秒	1.37 秒	0.96 倍

E 型変換処理手続きの高コスト手続き内に、高コストループが 2 つ含まれる。これらのループ以外では、大きな性能差は見られなかった。(1.3 倍)

表 6. write ライブラリコスト比較 2

	FX1(クロック比補正)	PRIMERGY RX200S3	性能差
ループ 1	1.55 秒	0.73 秒	2.53 倍
ループ 2	1.29 秒	0.58 秒	2.65 倍
これ以外のループ	1.11 秒	0.85 秒	1.31 倍

性能差原因：

高コストループ 1，ループ 2 部分には、FX1 と PRIMERGY RX200S3 で、型変換の命令セットの違いによる以下の影響がある。

- FX1 では、r8 から i4 への型変換の際に、メモリへの store と load が発生する。
- PRIMERGY RX200S3 では､r8 から i4 への型変換の際に､メモリへの store と load が発生しない。

■FX1 r8→i4 型変換アセンブラ(例)

```
fdtoi    %f26,%f28
st       %f28,[%fp+1991]
ldsw     [%fp+1991],%g4
```

■PRIMERGY RX200S3 r8→i4 型変換アセンブラ(例)

```
cvttsd2si        %xmm3,%edx
```

図 13. real*8 write アセンブラ比較

注）FX1 では、「型変換→メモリへの store→メモリからの load」という処理になっている。

(2) real*8 read

real*8 read 処理の中で、ライブラリコストが大きいのは、10 進数部処理手続き、指数部処理手続きの 2 つ。その他の部分は、小さなコストの集まりであるため、調査対象から除外した。

表 7. read ライブラリコスト比較

	FX1(クロック比補正)	PRIMERGY RX200S3	性能差
10 進数部処理	1.49 秒	0.52 秒	2.83 倍
指数部処理	1.50 秒	0.50 秒	3.03 倍
これ以外の処理	3.00 秒	1.24 秒	2.42 倍

10 進数部処理手続きと指数部処理手続きについては、FX1 と PRIMERGY RX200S3 では switch 文の処理に違いがある。

- FX1 のライブラリは富士通 C コンパイラで翻訳されており、pic オプションを指定した時に、switch 文がジャンプテーブルを使用せず、case 文を一つずつ判定(if)し処理される。
- PRIMERGY RX200S3 の場合は、intel C コンパイラで翻訳されており、pic オプションを指定しても switch 文はジャンプテーブルを使用して処理されている。

※ pic オプション：位置独立コード(PIC) を生成することを指示するオプション

性能差原因：

アセンブラで確認すると、ジャンプテーブルを使用しない FX1 では、case の数だけ cmp 命令が生成されるため、命令数が増加している。

■switch文ソース（例）

```
for (i=0;  i < n; i++) {
    switch(値) {
    case  '0' : case  '1' : case  '2' : case  '3' : case  '4' :
    case  '5' : case  '6' : case  '7' : case  '8' : case  '9' :
              break ;
    case  ' ' :
              処理1
              break ;
    case  '-' :
              処理2
              return  4  ;
    case  '+' :
              処理3
              return  4  ;
    case  'e' : case  'd' : case  'q' :
    case  'E' : case  'D' : case  'Q' :
              処理4
              return  3  ;
    default :
              return  6  ;
    }
}
```

■FX1 アセンブラ（例）

```
.SSN153:
! 36355

        ldsb    [%o4],%o4
        cmp     %o4,48
        be,pt   %icc, .L1494
        nop

(省略：caseの数だけcmp命令が生成される。)

! 36355
        cmp     %o4,69
        be,pt   %icc, .L1492
        nop
! 36355
        cmp     %o4,68
        be,pt   %icc, .L1492
        nop
! 36355
        cmp     %o4,81
        be,pt   %icc, .L1492
        nop
```

■PRIMERGY RX200S3 アセンブラ(例)

```
# 24691
        jmp *   %rdx

以下のようなジャンプテーブルを生成し、使用している。

.LBT2:
        .quad   .LL1552
        .quad   .LL1563
        .quad   .LL1563
        .quad   .LL1563
        .quad   .LL1563
        .quad   .LL1563
        .quad   .LL1563
        .quad   .LL1563
        (省略)
        .quad   .LL1563
        .quad   .LL1563
        .quad   .LL1563
        .quad   .LL1563
        .quad   .LL1563
        .quad   .LL1563
        .quad   .LL1563
        .quad   .LL1580
```

図 14. real*8 read アセンブラ比較

(3) 性能差まとめ

性能差の原因として、FX1 では型変換処理（命令セットの違い）と switch 文の処理が原因であることが判明した。

型変換の改善策として、型変換をすることは処理上必要な部分であるため、ランタイムシステムライブラリのプログラム修正（型変換しない）による改善は難しい。しかし、命令密度を向上することにより、改善できないか検討している。

switch 文の処理については、pic オプション指定時でもジャンプテーブルを使用した処理をするように、コンパイラの改善を検討している。

以上

2.2.4. LMbench によるメモリレイテンシ測定

富士通株式会社　石附 茂

1. 概要

LMbench はマシンの基本性能を測定するツールである。測定項目は以下の 2 項目に大別される。

1) バンド幅
　メモリ，ファイル入出力関連
2) レイテンシ
　キャッシュ，メモリ，コンテキストスイッチ，ファイル操作，プロセス，シグナルなど

2. メモリレイテンシの実測

LMbench を使用し、メモリレイテンシを実測した結果を報告する。対象マシンは、富士通 RX200S3 である。レイテンシ測定機構および測定時の注意点などについて述べた後、結果を考察する。

2.1. レイテンシ測定機構

レイテンシの算出方法は、以下に示すアドレス更新処理を多数回実行することで行っている。

```
p = (char **)*p
```

レイテンシ時間＝測定時間 / 実行回数 により算出される。
なお、測定結果は、ナノ秒単位で標準出力に出力される。

2.2. レイテンシ測定時の注意事項

(1)ストライド幅の設定

指定されたストライド幅で配列にアクセスしており、指定値により測定結果は変化する。

・指定値が小さい → キャッシュラインの再利用効果で過小評価となる.
・指定値が大きい → TLB ミスの影響を計測するため過大評価となる.

よって、測定対象機のキャッシュラインサイズやキャッシュサイズを考慮し、適切な値を指定しなければならない。小サイズから大サイズまで複数パターン測定し、分析する必要がある。
「2.3.測定結果について」の節で、具体例を示し解説する。

(2)アクセス範囲の設定

指定されたサイズの範囲内でストライドアクセスする。指定するサイズが小さいと、キャッシュ部だけを評価することになる。つまり、キャッシュサイズ合計より大きい値を指定する必要がある。

(3)測定結果の分析

測定結果は、実際に計測した値が列挙されており、全ての値を比較しなければならない。
一般的には、L1$ 部，L2$ 部，メモリ部などの傾向が見えるはずである。しかし、全ての値を見るのは煩わしいので、グラフ化することを推奨する。L1$ 部，L2$ 部，メモリ部は平坦な領域となり視覚的に把握することができる。

2.3. 測定結果について

富士通 RX200S3（clovertown(2.66GHz)）における測定結果を 図-1 に示す。
グラフの縦軸はレイテンシ時間(n 秒)であり、横軸はアクセス範囲(MB)である。複数の曲線は、各 stride 幅(32, 64, 128, 256, 512, 1024, 2048, 4096)に対応している。ただし、128byte 以上は、値が同じであり曲線は重なっている。この結果から、以下の 2 点が読み取れる。

(1) stride 幅の影響

Stride 幅 32byte と 128byte では計測値が異なっている。32byte では stride 幅が小さいため、キャッシュライン再利用効果の影響により計測値が小さくなったと推測される。Stride 幅が 128byte

以上 4096byte までは、ほぼ同じ値となっている。ただし、これは clovertown の結果であり、他機種でも同様とは限らない。

(2)アクセスレイテンシ

グラフの形状から、3 ヶ所の平坦域が確認できる。これらは、一次キャッシュ，二次キャッシュおよびメインメモリに対応している。Stride 幅 512byte での計測結果は、

L1$：1.14 n 秒（≒3 cycle）
L2$：5.33 n 秒（≒14 cycle）
Mem：125　n 秒（≒332 cycle）

となっている。

3. 測定環境の構築方法

公開ホームページからベンチマークセットをダウンロードし、測定対象マシンにインストールする。

・http://lmbench.sourceforge.net から、「lmbench-3.0-a8.tgz」をダウンロードする。
・「lmbench-3.0-a8.tgz」を測定対象マシンに展開する。(C 言語コンパイラが必要)

4. 実行方法

下記のように make するだけで、コンパイルから測定まで自動的に実行される。

```
% cd lmbench-3.0-a8
% make results
```

ただし、途中でパラメータの入力が必要になる。(12 項目)

パラメータ入力例

```
MULTIPLE COPIES : [リターン]
Job placement selection : [リターン]
MB : [256] ← 確保するメモリ量を指定する（file I/O とメモリの両方で使用される）
SUBSET (ALL|HARWARE|OS|DEVELOPMENT) : [リターン]
FASTMEM : [リターン]
SLOWFS : [yes]
DISKS : [リターン]
REMOTE : [リターン]
Processor mhz : [リターン]
FSDIR : [/work] ← 測定対象ファイルシステム上のディレクトリを指定する
Status output file : [リターン]
Mail results : [no]
```

上記パラメータの入力が終わると、自動的に計測が開始され、標準出力に進行状況が表示される。

5. 実行結果の確認方法

1) テキスト形式でサマリーを出力する方法

ディレクトリ results に移動後、

```
% make summary
```

と入力すると、標準出力に測定結果のサマリーが表示される。
サマリーの出力例を文末の［参考資料］に示す。

2) 測定結果をグラフにする方法

ディレクトリ results に移動後、

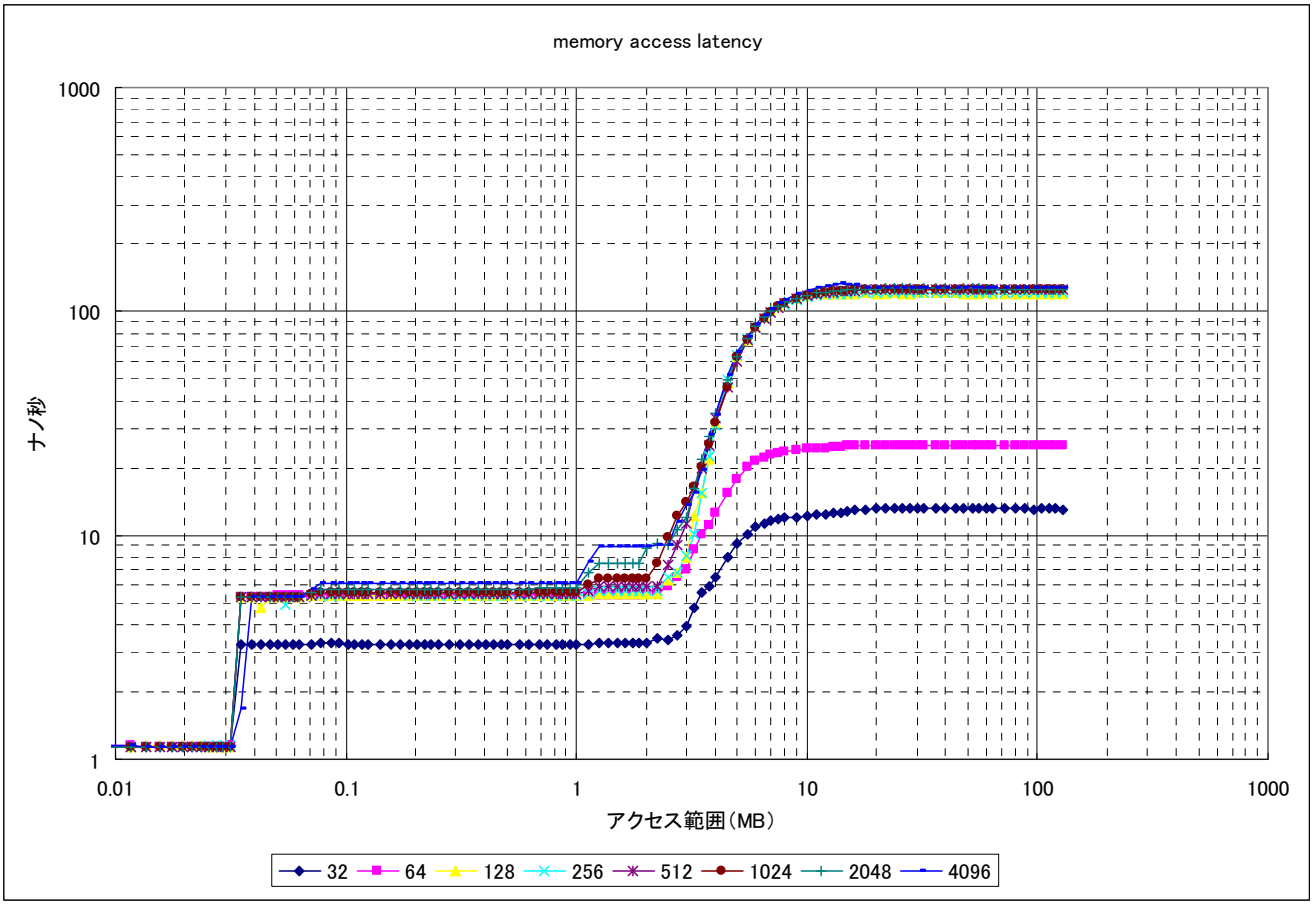
```
% make ps
```

と入力すると、測定結果をグラフにしたポストスクリプトファイルが作成される。
ポストスクリプトファイルの出力例を文末の［参考資料］に示す。

☆注意事項

ノード内に複数 CPU 存在する場合は、プロセスと CPU をバインドする必要がある。

バインドが困難な場合は、入力の「MULTIPLE COPIES」にノード内 CPU 数を指定することで、プロセスの CPU 間移動を抑える効果が期待できる。ただし、実行時間が極端に長くなる。



機種： RX200S3, Xeon X5355(clovertown 2.66GHz), FSB 1333MHz
L1$ ： 32 KB
L2$ ： 8 MB

図-1. メモリレンテンシ測定結果

以上

[参考資料]

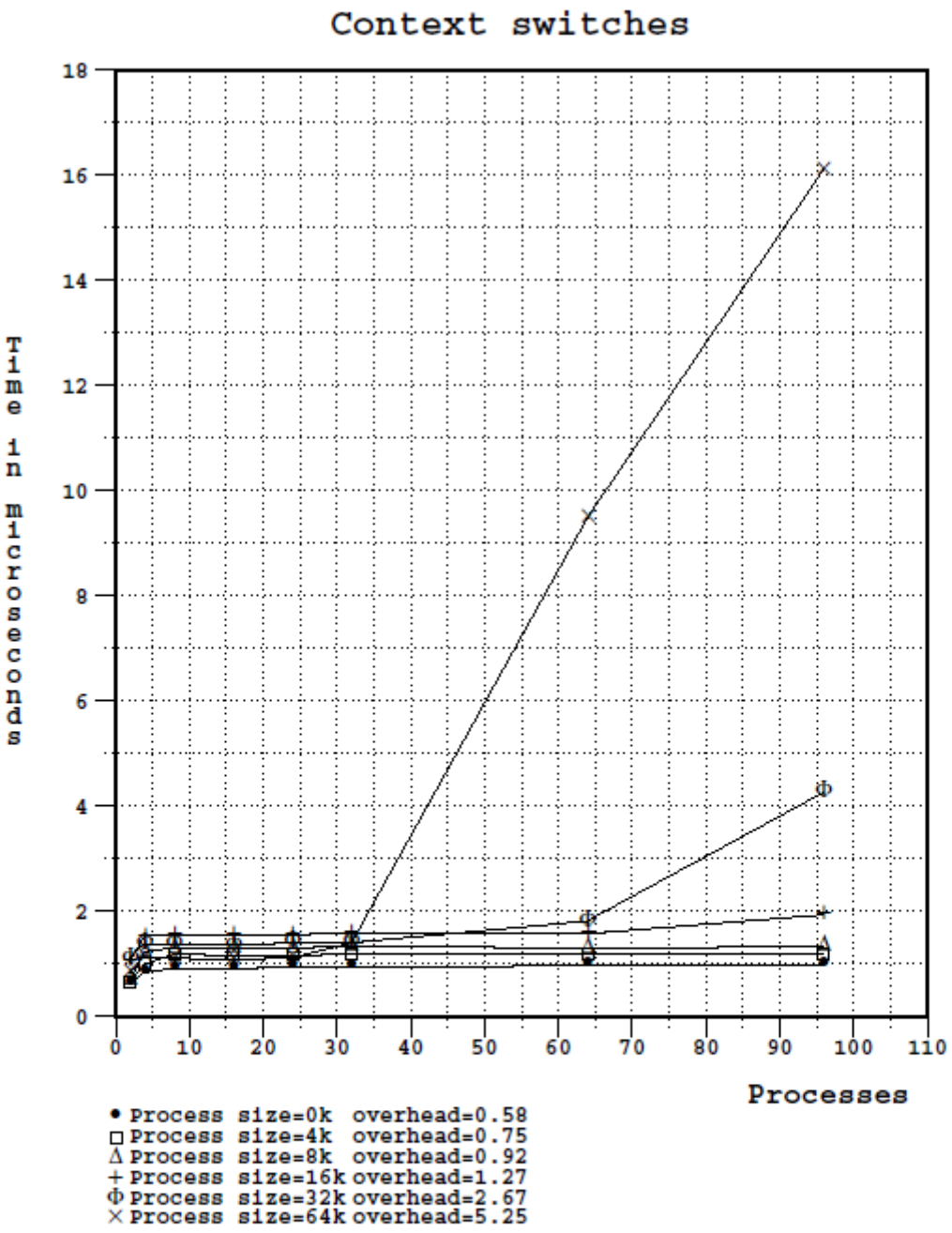
■サマリー出力例

```
                L M B E N C H  3 . 0   S U M M A R Y
                ------------------------------------
                (Alpha software, do not distribute)

Basic system parameters
----------------------------------------------------------------------------
Host                 OS Description              Mhz  tlb  cache  mem   scal
                                                     pages line   par   load
                                                           bytes
--------- ------------- ----------------------- ---- ----- ------ ------ ----
Machine           Linux               linux-gnu 2984     8   128 6.1200    1

Processor, Processes - times in microseconds - smaller is better
----------------------------------------------------------------------------
Host                 OS  Mhz null null      open slct sig  sig  fork exec sh
                             call  I/O stat clos TCP  inst hndl proc proc proc
--------- ------------- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
Machine           Linux 2984 0.09 0.14 2.58 2.88 8.16 0.18 1.28 82.0 261. 1051

Basic integer operations - times in nanoseconds - smaller is better
-------------------------------------------------------------------
Host                 OS  intgr intgr  intgr  intgr  intgr
                           bit   add    mul    div    mod
--------- ------------- ------ ------ ------ ------ ------
Machine           Linux 0.3400 0.1700 1.0200   11.3 6.8700

Basic uint64 operations - times in nanoseconds - smaller is better
------------------------------------------------------------------
Host                 OS int64  int64  int64  int64  int64
                           bit    add    mul    div    mod
--------- ------------- ------ ------ ------ ------ ------
Machine           Linux        0.1700

Basic float operations - times in nanoseconds - smaller is better
-----------------------------------------------------------------
Host                 OS  float  float  float  float
                           add    mul    div    bogo
--------- ------------- ------ ------ ------ ------
Machine           Linux 1.0200 1.3500 7.4800 5.8100

Basic double operations - times in nanoseconds - smaller is better
------------------------------------------------------------------
Host                 OS  double double double double
                           add    mul    div    bogo
--------- ------------- ------  ------ ------ ------
Machine           Linux 1.0200 1.6900   12.2   14.6

Context switching - times in microseconds - smaller is better
-------------------------------------------------------------------------
Host                 OS  2p/0K 2p/16K 2p/64K 8p/16K 8p/64K 16p/16K 16p/64K
                         ctxsw  ctxsw  ctxsw ctxsw  ctxsw   ctxsw   ctxsw
--------- ------------- ------ ------ ------ ------ ------ ------- -------
Machine           Linux 0.6100 1.0600 0.8300 1.5600 1.1000 1.54000 1.08000

*Local* Communication latencies in microseconds - smaller is better
---------------------------------------------------------------------
Host                 OS 2p/0K  Pipe AF     UDP  RPC/   TCP  RPC/ TCP
                        ctxsw       UNIX         UDP         TCP conn
--------- ------------- ----- ----- ---- ----- ----- ----- ----- ----
Machine           Linux 0.610 2.291 4.63 5.211 9.051 6.330  12.3 11.6

File & VM system latencies in microseconds - smaller is better
-------------------------------------------------------------------------------
Host                 OS   0K File      10K File     Mmap    Prot   Page   100fd
                        Create Delete Create Delete Latency Fault  Fault  selct
--------- ------------- ------ ------ ------ ------ ------- ----- ------- -----
Machine           Linux                              1509.0 0.446 0.99480 7.004

*Local* Communication bandwidths in MB/s - bigger is better
-----------------------------------------------------------------------
Host                 OS  Pipe AF    TCP  File   Mmap  Bcopy  Bcopy  Mem   Mem
                             UNIX      reread reread (libc) (hand) read write
--------- ------------- ---- ---- ---- ------ ------ ------ ------ ---- -----
Machine           Linux 1676 3383 1714 2378.4 3490.8 1434.5 1404.4 3215 1181.

Memory latencies in nanoseconds - smaller is better
    (WARNING - may not be correct, check graphs)
------------------------------------------------------------------------
Host                 OS   Mhz   L1 $   L2 $    Main mem    Rand mem    Guesses
--------- ------------- ---- ------ ------ -------- -------- -------
Machine           Linux  2984 1.0150 4.7580  111.2       125.3
```

■サマリー出力項目一覧

項目	サマリ内記述
CPU クロック (MHz)	Mhz
TLB エントリ数	tlb pages
キャッシュラインサイズ	cache line bytes
実行多重度	scal load
OS 呼び出しのレイテンシ	null call
read,write 関数の平均レイテンシ	null I/O
stat 関数のレイテンシ	stat
open,close の合計レイテンシ	open clos
ソケット read,write の合計レイテンシ	slct TCP
シグナル設定(sigemptyset)のレイテンシ	sig inst
シグナル送信(kill)のレイテンシ	sig hndl
fork のレイテンシ	fork proc
execv のレイテンシ	exec proc
sh 起動のレイテンシ	sh proc
整数の XOR 演算レイテンシ	intgr bit
整数の加算レイテンシ	intgr add
整数の乗算レイテンシ	intgr mul
整数の除算レイテンシ	intgr div
整数の剰余レイテンシ	intgr mod
64 ビット整数の XOR 演算レイテンシ	int64 bit
64 ビット整数の加算レイテンシ	int64 add
64 ビット整数の乗算レイテンシ	int64 mul
64 ビット整数の除算レイテンシ	int64 div
64 ビット整数の剰余レイテンシ	int64 mod
単精度実数の加算レイテンシ	float add
単精度実数の乗算レイテンシ	float mul
単精度実数の除算レイテンシ	float div
単精度実数四則演算{ a=(b+c)*(d-e)/f }のレイテンシ	float bogo
倍精度実数の加算レイテンシ	double add
倍精度実数の乗算レイテンシ	double mul
倍精度実数の除算レイテンシ	double div
倍精度実数四則演算{ a=(b+c)*(d-e)/f }のレイテンシ	double bogo
メモリ 0KB のプロセス 2 個のコンテキスト切り替え時間	2p/0K ctxsw
メモリ 16KB のプロセス 2 個のコンテキスト切り替え時間	2p/16K ctxsw
メモリ 64KB のプロセス 2 個のコンテキスト切り替え時間	2p/64K ctxsw
メモリ 16KB のプロセス 8 個のコンテキスト切り替え時間	8p/16K ctxsw
メモリ 64KB のプロセス 8 個のコンテキスト切り替え時間	8p/64K ctxsw
メモリ 16KB のプロセス 16 個のコンテキスト切り替え時間	16p/16K ctxsw
メモリ 64KB のプロセス 16 個のコンテキスト切り替え時間	16p/64K ctxsw
Pipe によるプロセス間待ち時間	Pipe
ソケットによるプロセス間通信待ち時間	AF UNIX
UDP/IP 経由のプロセス間通信待ち時間	UDP
UDP 経由の RPC の待ち時間	RPC/UDP
TCP/IP 経由のプロセス間通信待ち時間	TCP
TCP 経由の RPC の待ち時間	RPC/TCP
ソケットのコネクション時間	TCP conn
0KB のファイル生成時間	0K File Create
0KB のファイル削除時間	0K File Delete
10KB のファイル生成時間	10K File Create
10KB のファイル削除時間	10K File Delete
Mmap アクセスレイテンシ	Mmap Latency
Protection fault のレイテンシ	Prot Fault
Page fault のレイテンシ	Page Fault
100 個のデスクリプタに対する select システムコールの時間	100fd selct
ファイル読み込みを繰り返した時の平均バンド幅	File reread
Mmap アクセスを繰り返した時の平均バンド幅	Mmap reread
bcopy 関数のバンド幅	Bcopy (libc)
配列コピーのバンド幅	Bcopy (hand)
メモリ読み込みのバンド幅	Mem read
メモリ書き出しのバンド幅	Mem write
1 次キャッシュのアクセスレイテンシ	L1 $
2 次キャッシュのアクセスレイテンシ	L2 $
メモリのアクセスレイテンシ	Main mem
ストライド幅を変化させた場合のメモリアクセスレイテンシ	Rand mem

■context switch の出力例



コンテキストの切り替え時間

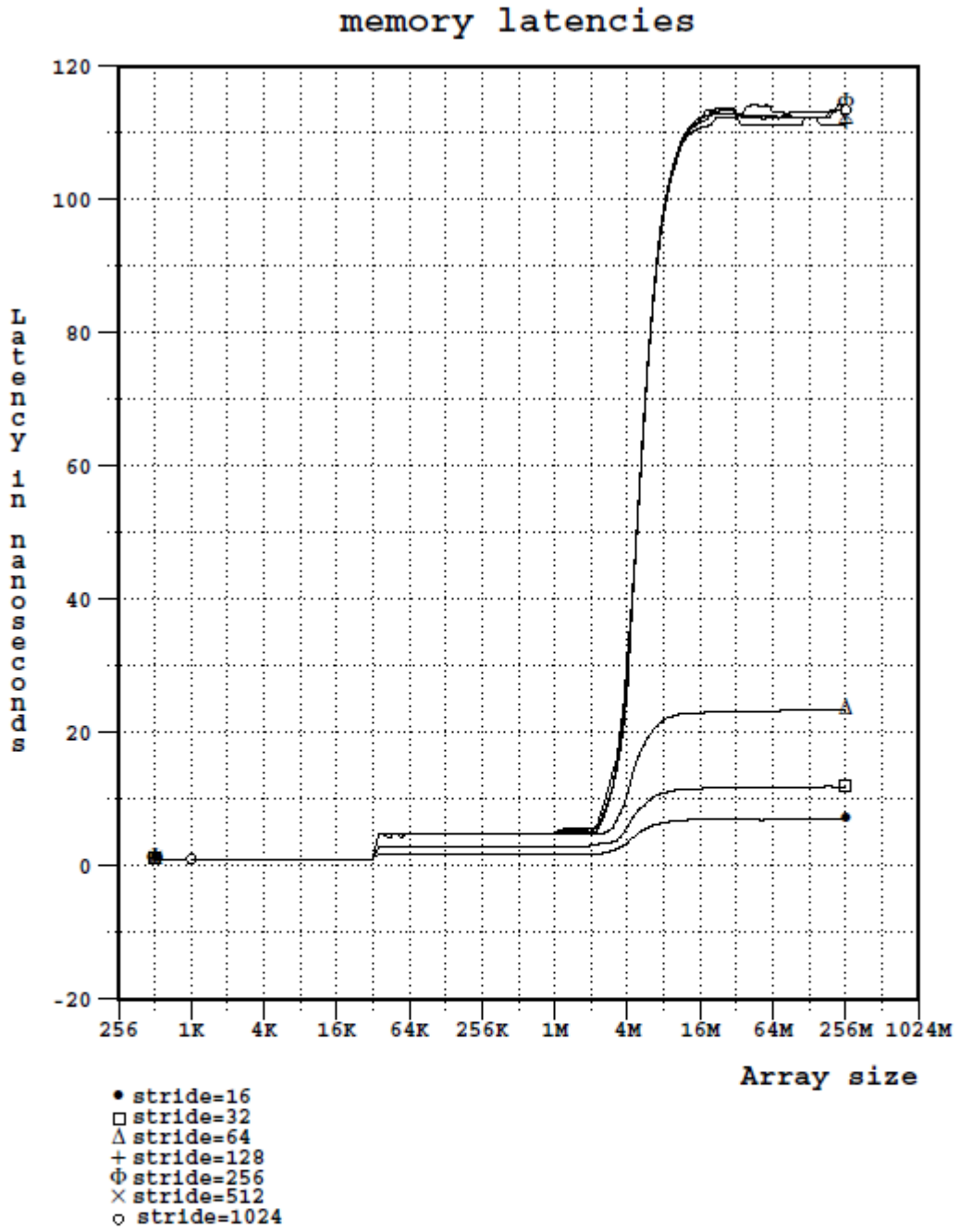
size はアロケートするメモリ量

overhead はメモリアロケート・配列定義参照などの処理時間

X 軸は生成(fork)するプロセス数

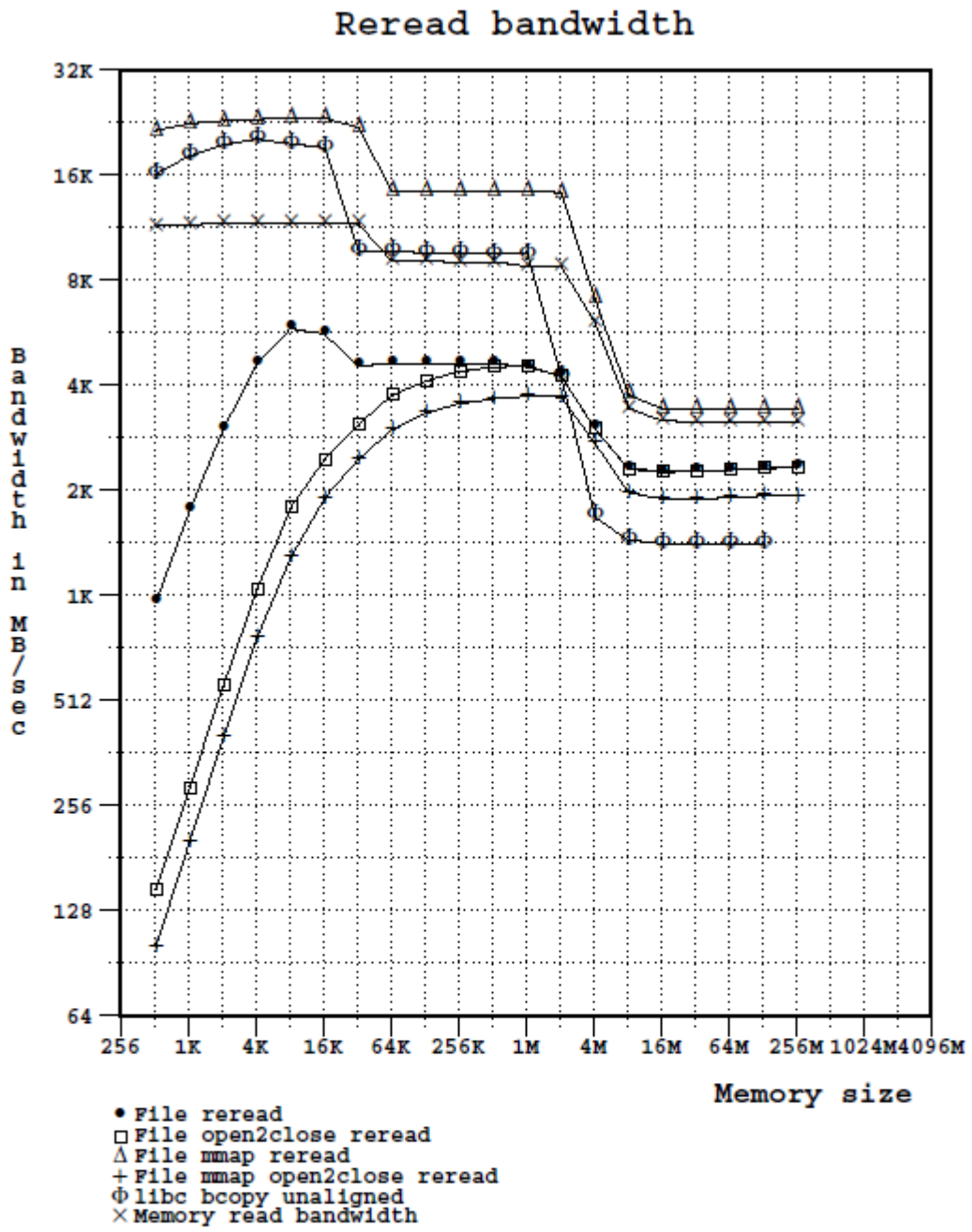
測定時間から overhead 時間を減算した値が評価値

■memory latency の出力例



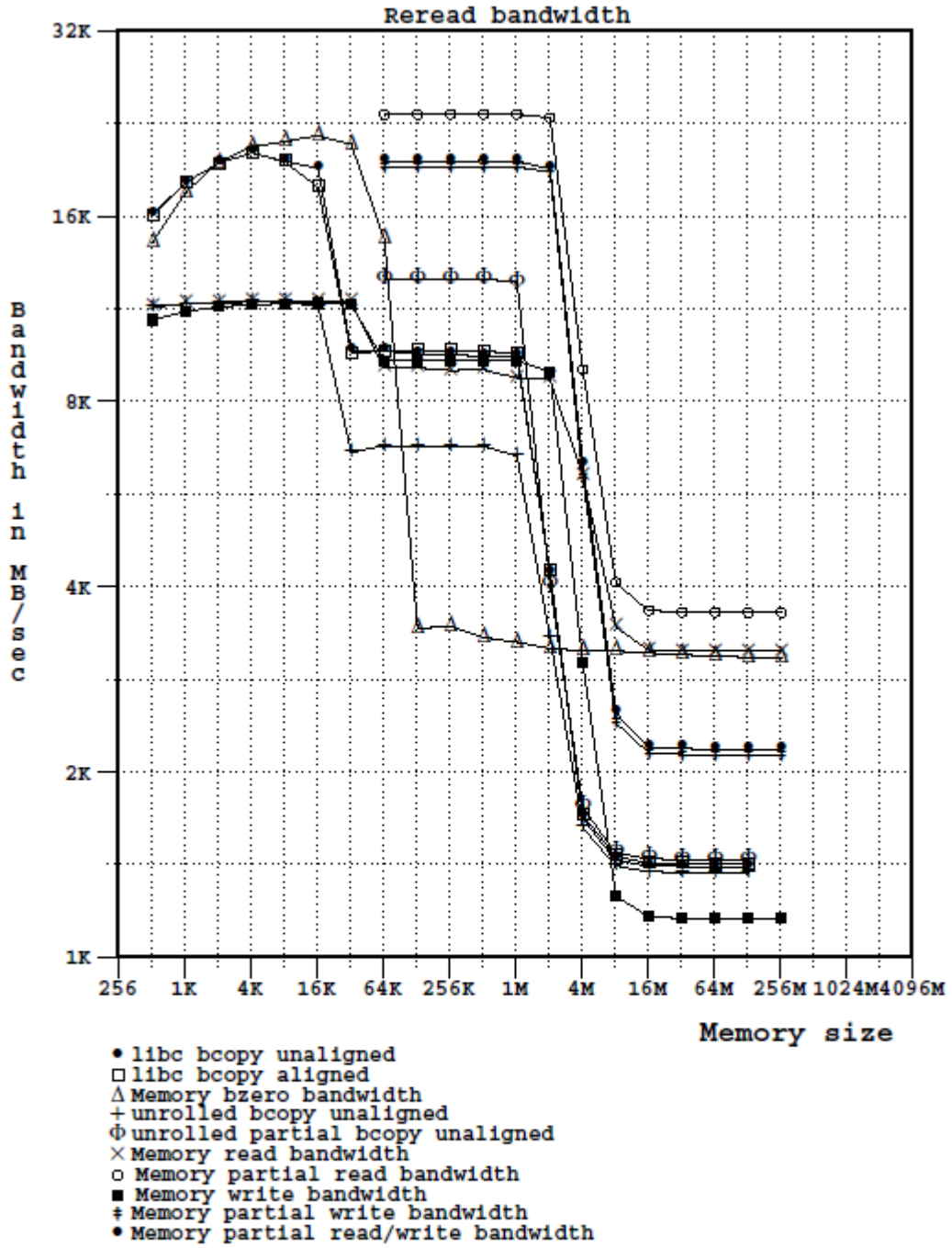
メモリアクセスのレイテンシ時間
stride はアクセス時の幅

■file read bandwidth の出力例



File reread : ファイルポインタを先頭に戻して read を繰り返す時間からバンド幅を計算
File open2close reread : open -> read -> close を繰り返す時間からバンド幅を計算
File mmap reread : mmap でマップ作成後、アクセスを繰り返す時間からバンド幅を計算
File mmap open2close reread : open -> mmap -> [access] -> close -> munmap を繰り返す時間
libc bcopy unaligned : bcopy を繰り返す時間からバンド幅を計算（8byte 境界ではない）
Memory read bandwidth : メモリ参照のバンド幅

■memory read bandwidth の出力例



libc bcopy unaligned : bcopy を繰り返す時間からバンド幅を計算（8byte 境界ではない）
libc bcopy aligned : bcopy を繰り返す時間からバンド幅を計算（8byte 境界）
Memory bzero bandwidth : bzero を繰り返す時間からバンド幅を計算
unrolled bcopy unaligned : 配列コピーを繰り返す時間からバンド幅を計算
unrolled partial bcopy unaligned : 32 バイトのストライドで配列コピーを繰り返す時間からバンド幅を計算
Memory read bandwidth : 連続メモリ参照のバンド幅
Memory partial read bandwidth : 32 バイトのストライドでメモリ参照のバンド幅
Memory write bandwidth : 連続メモリ定義のバンド幅
Memory partial write bandwidth : 32 バイトのストライドでメモリ定義のバンド幅
Memory partial read/write bandwidth : 32 バイトのストライドでメモリ参照&定義（同領域への総和）のバンド幅

2.3. アプリ性能

2.3.1. Intel クアッドコア CPU でのベンチマーク

東京海洋大学　　吉岡　諭

1. はじめに

この数年でマルチコア CPU の普及が進んできた。x86 系の CPU でも Intel と AMD がデュアルコア、クアッドコアの CPU を次々と市場に送り出していて、それらが PC クラスタの CPU として採用され、HPC に活用されている。ここでは Intel クアッドコア CPU を搭載した PC 単体で浮動小数点を中心としたベンチマークプログラムを実行し、CPU 及びコンパイラの並列性能を調べた結果を報告する。並列化には MPI、OpenMP、コンパイラの自動並列化機能を用いた。

2. ベンチマーク

今回ベンチマークプログラムとしては姫野ベンチマークを採用した。姫野ベンチマークがマルチコア CPU の性能を測定する最良のベンチマークということでは必ずしもないが、これまで日本で浮動小数点演算の実効性能を測定するベンチマークとして広く利用されてきたことを考慮し採用した。姫野ベンチマークは 3 次元格子でポアッソン方程式をヤコビの反復法で解く場合に主要となるループの処理速度を計るものである。その主要ループは以下の通りである。

```
    gosa= 0.0
    do k=2,kmax-1
       do j=2,jmax-1
          do i=2,imax-1
             s0=a(i,j,k,1)*p(i+1,j,k) &
                  +a(i,j,k,2)*p(i,j+1,k) &
                  +a(i,j,k,3)*p(i,j,k+1) &
                  +b(i,j,k,1)*(p(i+1,j+1,k)-p(i+1,j-1,k) &
                               -p(i-1,j+1,k)+p(i-1,j-1,k)) &
                  +b(i,j,k,2)*(p(i,j+1,k+1)-p(i,j-1,k+1) &
                               -p(i,j+1,k-1)+p(i,j-1,k-1)) &
                  +b(i,j,k,3)*(p(i+1,j,k+1)-p(i-1,j,k+1) &
                               -p(i+1,j,k-1)+p(i-1,j,k-1)) &
                  +c(i,j,k,1)*p(i-1,j,k) &
                  +c(i,j,k,2)*p(i,j-1,k) &
                  +c(i,j,k,3)*p(i,j,k-1)+wrk1(i,j,k)
             ss=(s0*a(i,j,k,4)-p(i,j,k))*bnd(i,j,k)
             gosa=gosa+ss*ss
             wrk2(i,j,k)=p(i,j,k)+omega *ss
          enddo
       enddo
```

測定に用いたプラットホームは以下のものである。

(1) Intel Core 2 Extreme QX6700 (4 コア)

model name :　　Intel(R) Core2 Quad CPU　(Kentsfield) 2.66GHz
L2 cache size :　4096 KB×2
FSB :　　1066MHz
OS :　　CentOS 5.0 for Intel64

(2) Intel Xeon　E5462　2CPU (4x2 コア)

model name :　　Intel(R) Xeon E5462 Quad CPU　(Harpertown)　2.80GHz
L2 cache size :　6MB× 2/cpu
FSB :　　1600MHz
OS :　　Fedora 8 for Intel64

(3) Intel Core i7 940　(4 コア)

model name :　Intel(R) Core i7 Quad CPU　(Nehalem)　2.93GHz
L2 cache size :　256kB/core
L3 cache size :　8MB (共有)
QPI :　4.8GHz
OS :　Cent OS 5.2 for Intel64

なお、STREAM ベンチマークを用いて測定した両システムのメモリバンド幅は以下の通りである。

(1) QX6700 :1 コアでは 4.6GB/sー4.7GB/s、4 コア(openmp)でも 4.6GB/sー4.7GB/s
(2) E5462 :1 コアでは 4.7GB/sー5.5GB/s、8 コア(openmp)では 8.1GB/sー8.8GB/s
(3) Core i7 940:1 コアでは 5.7GB/sー10GB/s、4 コア(openmp)では 11GB/sー16GB/s

測定に用いたコンパイラは以下のものである。

A) Fujitsu Fortran Version 3.0
B) Intel Fortran Compliler 10.0.026 (Core i7 940 だけ 11.1.056)
C) GNU gfortran 4.1.2

3. 測定結果

ベンチマークの測定は 4 つの格子サイズで行った。

XS (64x32x32)
S (128x64x64)
M (256x128x128)
L (512x256x256)

利用する配列の総バイト数は、XS では 3.6MB、S では 29MB、M では 235MB、L では 1.9GB になる。
測定結果を以下に示す。

3.1 スカラー性能

コンパイラオプションは以下の通りである。

gfortran -O3 (GNU)、ifort -O3 (Intel)、 frt -Kfast (Fujitsu)

測定結果を図 1 に示す。どのシステムとも格子サイズが大きくなるに従って、性能が落ちている。特に XS と S の間で性能差が大きい。XS では全配列が L2 キャッシュにおさまるのに対して、S 以上の格子サイズでは配列が L2 キャッシュから溢れていることが原因と考えられる。また、Core i7 940 (Nehalem)のスカラー性能が高いことが目立つ。

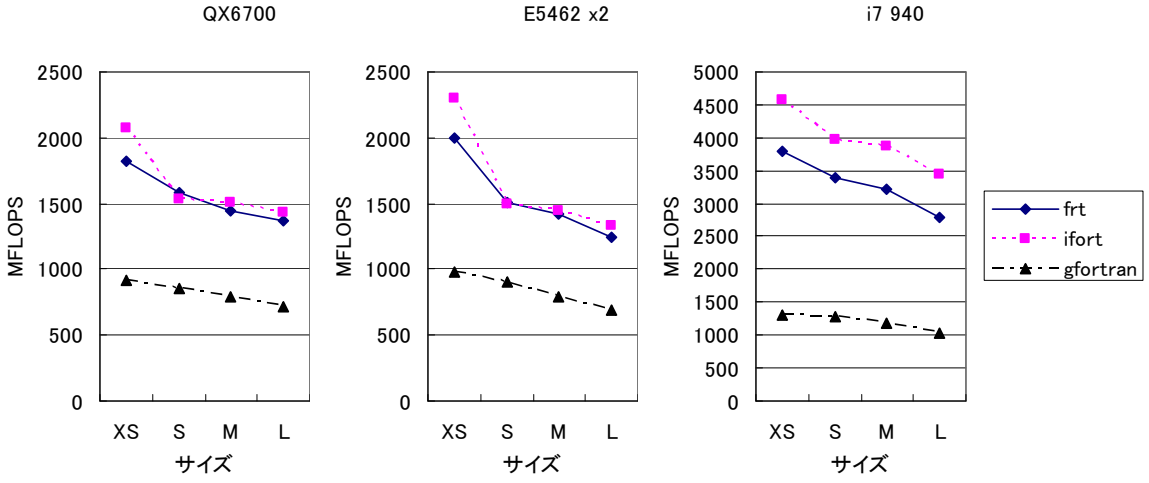


図 1. スカラー性能

3.2 自動並列化

GNU gfortran は自動並列化機能を持っていない。

スカラー版のベンチマークコードはそのままでは Intel コンパイラ ver10.0 及び ver11.1 では自動並列化されなかった。そのため、わずかなコードの変更とコンパイルオプションの追加を行うことによって、自動並列化を行った。

コンパイラオプションは以下の通りである。

ifort -O3 -parallel -par-threshold99 (Intel)、 frt -Kfast,parallel (Fujitsu)

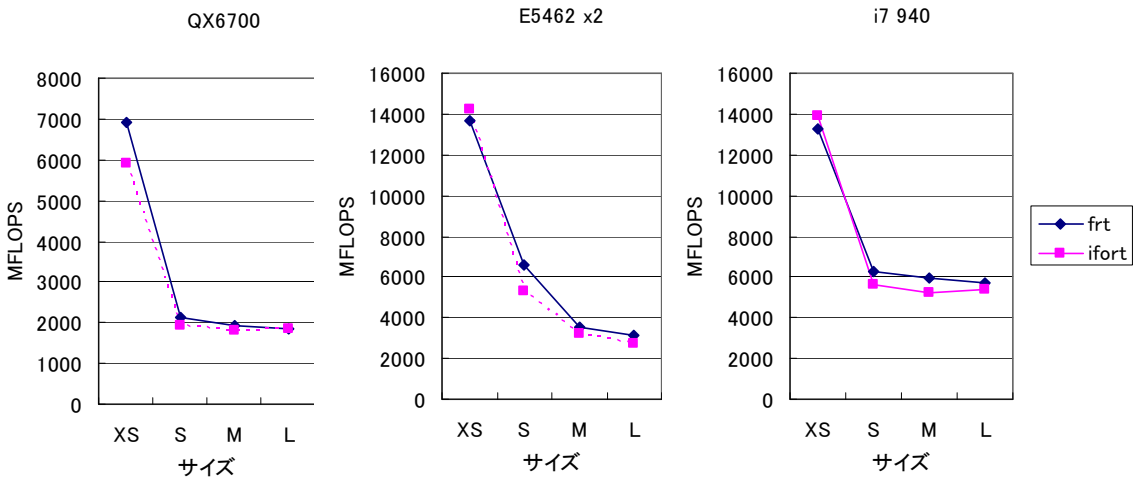


図 2. 自動並列化性能

測定結果を図 2 に示す。ここでも各システムとも格子サイズが大きくなるに従って、性能が落ちている。スカラー計算に対する性能向上率は富士通コンパイラでは、QX6700 で 1.35 倍(L サイズ)から 3.78 倍(XS サイズ)、E5462 で 2.52 倍(L サイズ)から 6.85 倍(XS サイズ)、i7 940 で 2.07 倍(L サイズ)から 3.51 倍(XS サイズ)となっている。格子サイズ XS では性能向上率がかなり高く、L2 キャッシュにデータがおさまっている場合には、マルチコアの威力が発揮できているようである。 Core i7 940 はコア数が 4 つであるのに、その倍のコア数の E5462x2 と同等以上の性能を示している。

3.3 OpenMP

コンパイラオプションは以下の通りである。

gfortran -O3 -fopenmp (GNU)、ifort -O3 -openmp (Intel)、 frt -Kfast,OMP (Fujitsu)

測定結果を図 3 に示す。ここでも各システムとも格子サイズが大きくなるに従って、性能が落ちている。GNU コンパイラの OpenMP 性能はかなり低いこともわかる。スカラー計算に対する性能向上率は富士通コンパイラでは、QX6700 で 1.36 倍(L サイズ)から 3.85 倍(XS サイズ)、E5462 で 2.40 倍(L サイズ)から 6.66 倍(XS サイズ)、i7 940 で 2.06 倍(L サイズ)から 3.50 倍(XS サイズ)となっている。自動並列化と同様に、格子サイズ XS では性能向上率がかなり高い。自動並列の場合と同様に Core i7 940 は E5462x2 と同等以上の性能を示している。

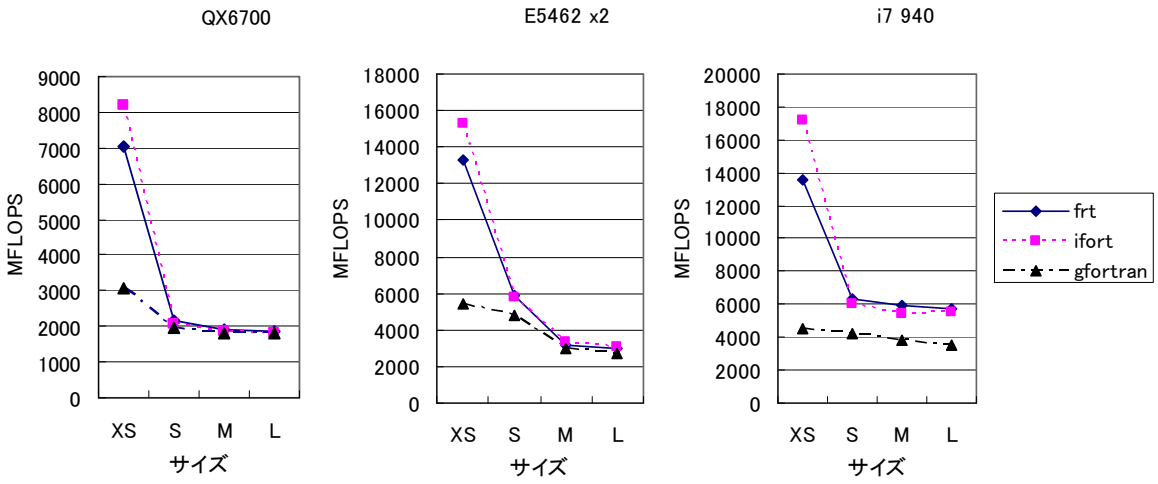


図 3. OpenMP 性能

3.4 MPI

MPI ライブラリは OpenMPI 1.2.4 を用いた。1 次元方向の領域分割を用いて並列化をしている。

コンパイラオプションは以下の通りである。

gfortran -O3 （GNU）、ifort -O3 （Intel）、 frt -Kfast （Fujitsu）

測定結果を図 4 に示す。各システムとも格子サイズが大ききなるに従って、性能が落ちている。スカラー計算に対する性能向上率は富士通コンパイラでは、QX6700 で 1.33 倍（L サイズ）から 2.79 倍（XS サイズ）、E5462 で 2.00 倍（L サイズ）から 6.42 倍（XS サイズ）、i7 940 で 2.23 倍（L サイズ）から 3.51 倍（XS サイズ）となっている。性能向上率は他の並列計算に比べてわずかに小さい。

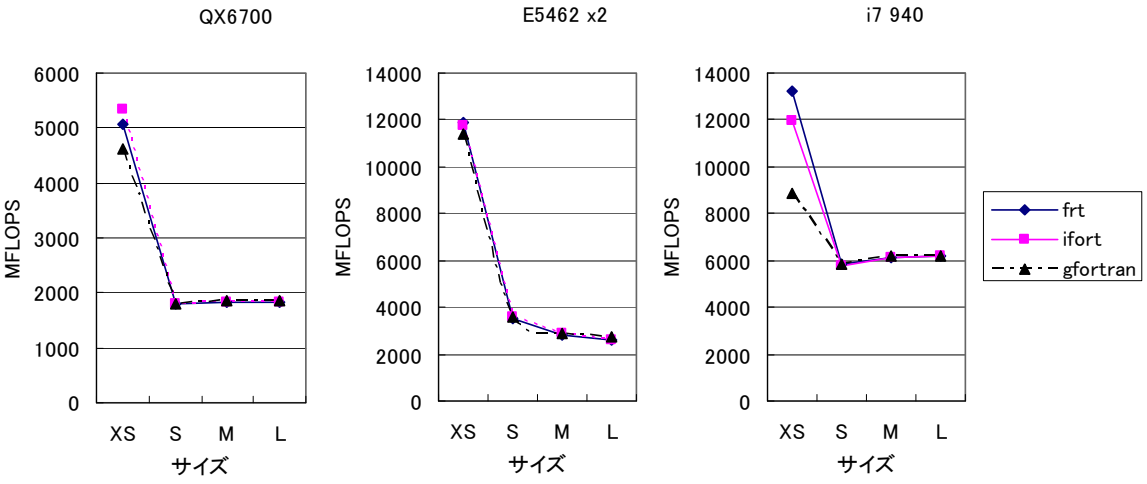


図 4. MPI 性能

なお、すべての並列計算で、QX6700 では S～L サイズで同等の結果なのに対して、E5462 では S サイズの性能が M、L サイズの性能を大きく上回っている。その理由として、利用できる L2 キャッシュの総量が QX6700 では 8MB なのに対して、E5462 では 24MB あり、これは S サイズの計算が必要とするメモリ量にほぼ匹敵し、ある程度 L2 キャッシュを利用した計算ができているためと考えられる。

Core i7 940 は L サイズの計算で E5462x2 の倍の性能を示している。

3.5 コア数と並列性能

E5462 で富士通コンパイラを用いて、利用するコア数（スレッド数）に対する性能測定を自動並列、OpenMP、

MPI それぞれに対して測定した結果を図 5 に示す。基本的には利用するコア数が増すにつれて性能は上がっているが、MPI では 8 コアの性能が 4 コアの性能を下回る場合がある。

4. 考察

4.1 コンパイラ

スカラ性能は富士通コンパイラとインテルコンパイラ ver10.0&Ver11.1 が高い。並列計算した場合ではコンパイラによる性能差はあまりない。マルチコアで計算したからといってコア数分計算が速くなる訳ではない。ベンチマークのサイズが大きい場合には、MPI、OpenMP、コンパイラの自動並列化機能、いずれの並列化を適用した場合にも性能はあまり変わらない。そういう意味では コンパイラの自動並列化機能は健闘していると言える。

4.2 メモリバンド幅とキャッシュ

シングルコアの CPU では、キャッシュを活用した計算が出来るかどうかが性能を左右していた。その状況はマルチコア CPU でも変わらない。マルチコア化によって CPU 全体としての演算性能が上がったため、メモリバンド幅に対する要求が大きくなっていて、結果としてキャッシュを活用できるかどうかが性能に大きく影響を与えている。特にここで測定した姫野ベンチマークのように、浮動小数点演算数に対する、メモリへのアクセス数の比率が高い場合には、その影響が大きい。ただし、マルチコア化によって、結果的に1CPU あたりで利用できるキャッシュの量が増えたため、シングルコアではキャッシュに載らなかった計算がマルチコアではキャッシュにおさまるようになり、性能が上がるということもある。

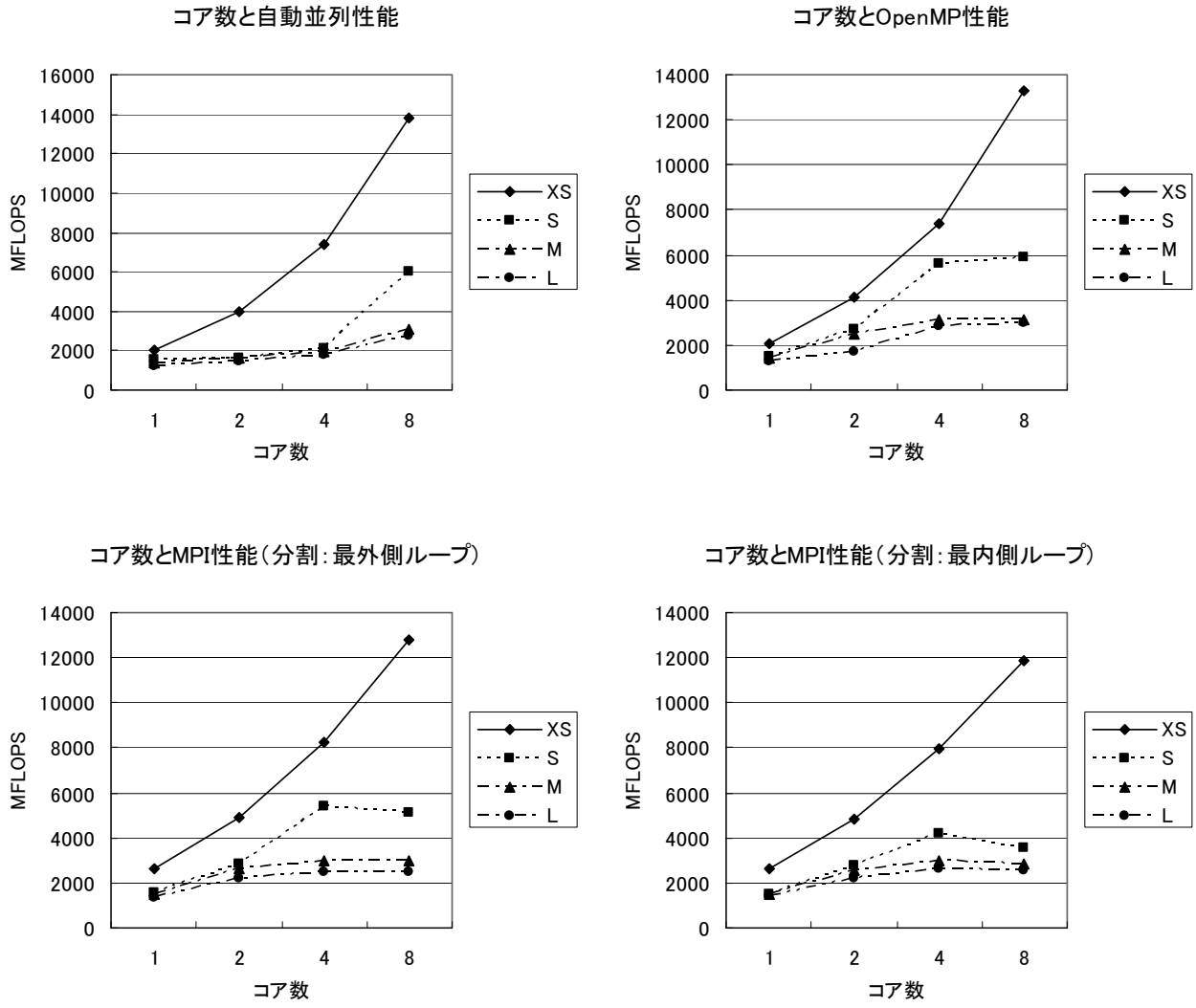


図 5. コア数と並列性能

以上

【補足資料】

■QX6700 のベンチマーク結果：サイズと MFLOPS 値（スカラー以外は 4 コア）

スカラー

サイズ	frt	ifort	gfortran
XS	1825.4	2076.06	914.64
S	1580.27	1536.02	859.49
M	1440.79	1509.78	787.01
L	1369.46	1429.59	710.71

自動並列

サイズ	frt	ifort
XS	6916.59	5903.31
S	2127.82	1914.47
M	1919.27	1819.46
L	1860.56	1837.81

OpenMP

サイズ	frt	ifort	gfortran
XS	7021.03	8213.53	3085.21
S	2141.4	2050.36	1975.97
M	1914.78	1874.47	1834.03
L	1859.85	1831.23	1790.23

MPI

サイズ	frt	ifort	gfortran
XS	5083.88	5332.59	4612.03
S	1803.73	1799.86	1792.8
M	1826.73	1827.02	1848.51
L	1825.07	1835.12	1854.96

■E5462 x2 でのベンチマーク結果：サイズと MFLOPS 値（スカラー以外は 8 コア）

スカラー

サイズ	frt	ifort	gfortran
XS	1993.35	2305.17	985.71
S	1512.17	1500.79	899.22
M	1415.78	1442.64	787.19
L	1247.71	1333.44	694.25

自動並列

サイズ	frt	ifort
XS	13653.96	14231.42
S	6570.77	5289.95
M	3550.24	3182.46
L	3140.34	2698.68

OpenMP

サイズ	frt	ifort	gfortran
XS	13285.28	15269.24	5462.88
S	5889.66	5795.5	4790.59
M	3159.33	3325.99	3020.31
L	2999.57	3068.13	2752.87

MPI

サイズ	frt	ifort	gfortran
XS	11897.21	11735.14	11393.37
S	3538.89	3568.11	3601.4
M	2826.37	2855.61	2851.65
L	2588.95	2625.89	2738.42

■Core i7 の 940 ベンチマーク結果：サイズと MFLOPS 値（スカラー以外は 4 コア）

スカラー

サイズ	frt	ifort	gfortran
XS	3784.19	4564.07	1295.31
S	3393.25	3982.34	1277.67
M	3220.94	3859.39	1181.88
L	2779.55	3431.87	1018.58

自動並列

サイズ	frt	ifort
XS	13279.37	13936.09
S	6295.6	5630.32
M	5968.83	5254.35
L	5745.72	5354.06

OpenMP

サイズ	frt	ifort	gfortran
XS	13571.87	17146.44	4535.93
S	6313.61	6013.17	4177.21
M	5935.83	5465.88	3849.17
L	5738.53	5498.23	3480.91

MPI

サイズ	frt	ifort	gfortran
XS	13252.81	11939.04	8861.22
S	5805.04	5801.21	5811.25
M	6126.84	6131.4	6158.61
L	6196.14	6193.41	6220.98

■E5462 x 2 でのベンチマーク結果：コア数と MFLOPS 値

自動並列

コア数	parallel (XS)	parallel (S)	parallel (M)	parallel (L)
1	2001.83	1513.36	1414	1217.98
2	3954.33	1617.49	1611.3	1488.37
4	7363	2112.21	1934.28	1769.42
8	13837.51	6036.84	3121.32	2758.49

OpenMP

コア数	OpenMP (XS)	OpenMP (S)	OpenMP (M)	OpenMP (L)
1	2073.09	1510.62	1414.12	1252.38
2	4089.02	2705.88	2517.85	1721.09
4	7376.55	5604.05	3159.14	2837.45
8	13285.28	5889.66	3159.33	2999.57

MPI
（分割は最外側ループ）

コア数	MPI (XS)	MPI (S)	MPI (M)	MPI (L)
1	2662.82	1542.29	1512.72	1358.4
2	4906.28	2843.86	2627.57	2207.25
4	8220.93	5394.65	2964.1	2497.33
8	12798.31	5134.03	2993.13	2491.56

MPI
（分割は最内側ループ）

コア数	MPI (XS)	MPI (S)	MPI (M)	MPI (L)
1	2661.75	1518.47	1514.34	1389.72
2	4861.83	2792.92	2584.6	2169.11
4	7934.51	4212.4	2950.75	2622.96
8	11895.4	3538.89	2817.83	2588.95

2.3.2. 半古典分子動力学計算を用いたコンピュータ性能の測定

上智大学　　南部 伸孝

1. 概要

古典力学を基に分子の運動（粒子の運動）を記述する分子動力学シミュレーション（Molecular Dynamics simulation）が生体関連の分野で頻繁に利用され，最近では分子機械の解明等までも利用されている。特にそこで活躍されている数値計算法として速度ベルレ（Velocity-Verlet）法があるが，エネルギーの誤差が比較的大きい。一方，分子機能自体が量子現象に起因する場合もある。その場合は，分子の運動を古典論ではなく，量子論あるいは半古典論に基づき数値的に求めなければならない。ところが，計算精度の低い速度ベルレ法では破たんするため，より高精度な積分法を用いる必要が出てくる。

本計測では，ベンチマークプログラムとして古典トラジェクトリを半古典論に基づく経路積分法により量子効果を取り入れるため，Gray らが提唱する高精度な 4 次のシンプレクティック積分法［Brewer, Hulme, Manolopoulos, *J. Chem. Phys.* **106**, 4832-4839 (1996)の Appendix を参照］を用いた方法で実施した。また，【補足資料】(1)に実際のコードの一部を添付する。

2. 測定

測定を実施した計算機の CPU 型番は以下の 9 種類である。Intel と AMD と IBM が入り乱れた順番となっているが，リリースされた年代に沿って順番を振っていることから，年代とともに見ていただきたい。測定は 3 回実施し，その平均値を値とした。

①AMD Athlon™ 64 Processor 3500+
②AMD Athlon™ 64 X2 Dual Core Processor 4400+
③Intel® Pentium® 4 3.4GHz
④Intel® Xeon® 5160 3.0GHz（富士通 Primergy, PG）
⑤IBM Power5 1.9GHz （日立 SR11000 J1 ノード）
⑥Intel® Itanium2-p9000 1.6GHz（富士通 Primequest, PQ）
⑦Quad-Core AMD Opteron™ Processor 2346 HE @ 1.8GHz
⑧Intel® Core™2 Duo CPU E6850 @ 3.00GHz
⑨Dual-Core AMD Opteron™ Processor 2214 HE @ 2.2GHz

また，使用したコンパイラは以下の通りである。

1. 富士通製 Fujitsu Fortran Driver Version 2.0 P-id: T05097-03 (Sep. 5 2007 17:37:52)
2. 日立製最適化 FORTRAN90 V01-05
3. Intel® Fortran Compiler for Intel® EM64T-based applications, Version 9.1
4. The Portland Group, Inc. pgf90 6.0-4 64-bit target on x86-64 Linux

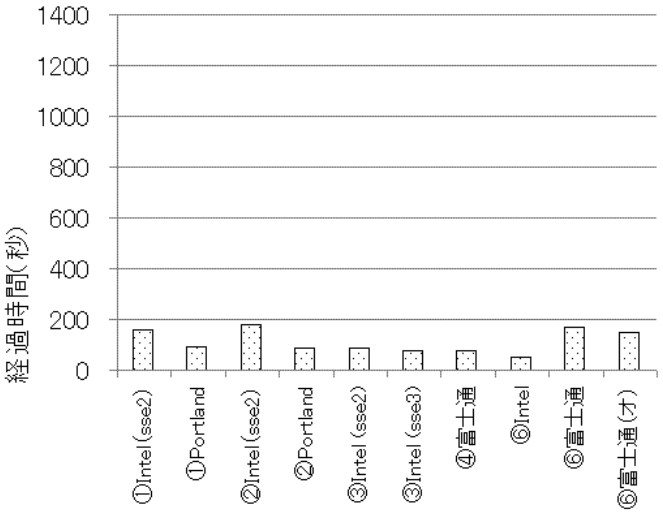


図 1. 非並列コンパイラの結果

3. 結果と考察

詳細なコンパイラオプション，計測経過時間等は【補足資料】(2)に列挙するので，それを参照されたい。ここでは，幾

つか特徴的に部分を取り上げて，結果を紹介する。まず，非並列コンパイルの結果を図 1 に示す。横軸は，使用したコンパイラのメーカーであり，例えば「⑥富士通(オ)」は富士通のコンパイルを使うが，オプションを最適化したものである。

具体的な結果であるが，非非並コンパイルと非並列実行の基，最も計算時間の短かったのは，Itanium2 の CPU 上で Intel コンパイラを用いた場合であった。50 秒弱であり，並列計算結果を含めても 2 番目である。②は AMD であるが，③の Intel との差異があまり見られない。但し，AMD では Portland 製を利用した方が良さそうである。一方,③と④はともに Intel の CPU であり混ぜて比較すると，型番が違うので明言はできないが，コンパイラの性能において富士通 vs. Intel は大差なしと思われる。

次に，それぞれのマシンを固定し，コンパイラ性能を比較する。図 2 は AMD Athlon の Dual Core の CPU である。東工大がみんなのスパコンとして導入した CPU より古い CPU であるが，結果の通り全く並列性能向上が見られない。その一方，Intel を除き，悪化も見られず，その性能を保持している。多分，オプションを選んでも全く並列化されなかったためだと思われる。

図 3 は Intel Xeon（別名：Nehalem コア）上での性能である。特に 1 ノード，4 コアまで共有メモリー型のマシンであることから，4 コアまで並列性能が期待できる。結果は，ご覧の通り，スレッド数が増えると悪化するのが分かる。そこで，最適オプションを富士通さんに選んで頂いくことにする。図 2 の結果と同様，自動並列を止める方向に動いていることがわかる。残念であるが，利用者からみた場合，悪化しなくなることも重要な要素なので，このコードでは，自動並列が期待できないと考えるべきかもしれない。

図 4 は日立 SR11000 上での性能である。もちろん, CPU は IBM Power5 であるが，4 コアで最短の 38.550 秒を記録した。ハードとソフトの同調性が見られる。日本は，ソフトウェアの開発において昔から才能がないようなことを言われているが，ゲームソフトとこのコンパイラの性能は，日本が誇るソフトウェアと自負すべきである。（富士通さん，頑張ってください！

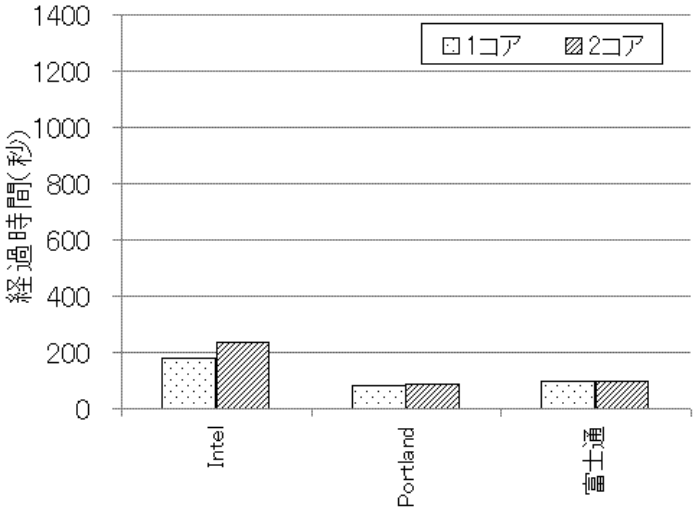


図 2. AMD 上における並列コンパイラ性能

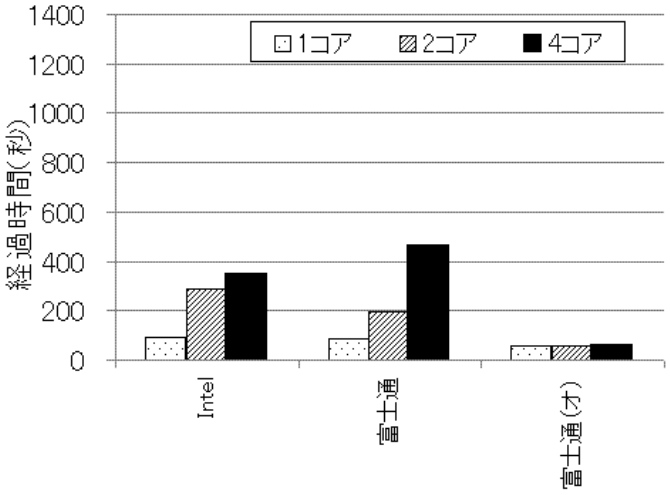


図 3. Intel Xeon 上における並列コンパイラ性能

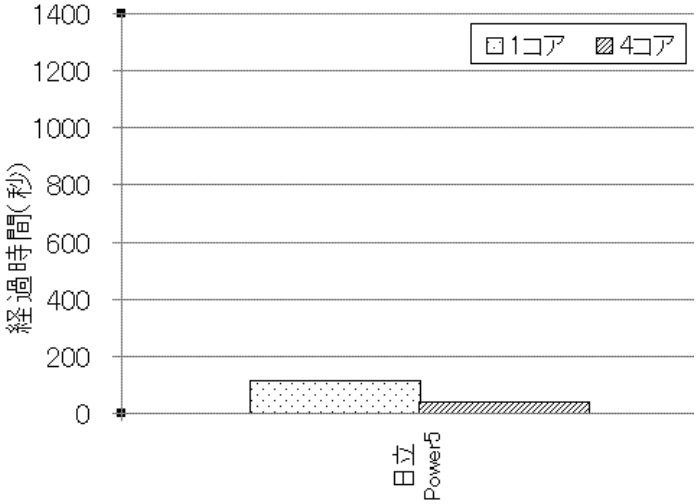


図 4. 日立 SR11000 上における並列コンパイラ性能

お願いします。また，可能だと期待しております。）

図 5 は Intel Intanium2 上での性能である。特に 1 ノード，64 コアまでの共有メモリー型マシンであることから，64 コアまで並列性能が期待できる。（気をつけなければならないことは,使用した計算機が SGI 社製 Altix 等とは異なり，物理的に共有メモリー型のマシンである。つまり，論理型の共有メモリー型マシンではない。）傾向は明らかに図 3 と同じである。並列性能が全く期待できない。そして，図 3 と比較するとかなり悪化する。無理に，共有メモリー型マシンを作成したのかもしれない。また上述と同様に，富士通の最適なオプションを選ぶと,悪化しなくなったが，性能の向上は見られない。

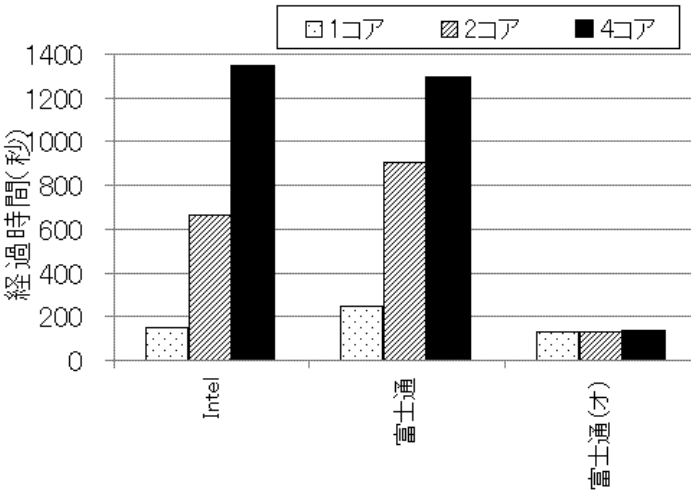


図 5. Intel Itanium2 上における並列コンパイラ性能

その後，⑦，⑧，⑨のマシンでも富士通の最適なオプションを選べば，悪化せず。Intel は悪化し，Portland は富士通の最適なオプションと同様な傾向を示す結果となった。

最後に，自動並列化という視点でのハイライトを図 6（本文の最後に掲載）に示す。

4. まとめ

- 最短だったのは，日立コンパイル&SR11000 モデル J1 上で，4CPUcore を用い並列実行した結果（38.550 秒）であった。1CPUcore の時が 113.329 秒から考えると自動並列化が機能しているように感じられる。
- 2 番目に短かったのは，49.978 秒を記録した Intel コンパイラ Version 9.1 を用いた非並列コンパイル&富士通 Primequest 上での非並列実行であった。この値には少し驚いている。何故だろうか？
- 富士通さんが新たに行ったチューニングのうちオプションのみ使い再計測を行った。PG では約 30%，PQ では約 50%（ただし，並列実行可能なバージョンを用いたとき）の性能向上がみられた。また，どちらも並列実行時に性能向上が見られないが，悪化がなくなった。（AMD バルセロナでもそうかもしれない。）一方，以前のオプションでは悪化が見られた。オプションの説明をお願いしたい。
- 日立コンパイラ以外，自動並列性能がかなり悪いことが分かる．Many cores の時代が間近に迫っていることを考えると早急の対応が求められる。
- AMD のバルセロナは，富士通さんのコンパイラだと Portland Group を抜いているが，デュアルコアのオプテロンやアスロン 64X2 だと遅くなる。特異な命令を使っているのだろうか？　一方，512K のキャッシュは少なすぎる。Molpro を使った大行列の固有値問題では話にならない。Gaussian でも同様，困ったものだ。

5. 謝辞

これまで約 16 年間以上，分子科学の分野においてスーパーコンピュータの管理および調達に携わってきた。振り返ると，日立 S820, M682 (2CPU) から始まり, SR2201, SR8000, SR11000, SR16000, NEC SX-3, SX-4R, SX-5, SX-7, 富士通 VPP5000, PrimeQuest, IBM SP2, SGI Origin 2000, Origin2800, Altix3700, Altix4700 等まで相手に奮闘してきた気がする。余談だが，UNIX システムは DEC VAX-11/750, SONY NEWS 830 から利用し，恐らく日本で初めてインターネットを利用した研

究者の一人だと思う。朴さんには学生時代とてもお世話になった。そして縁があり，平成 21 年の春，上智大学 理工学部 物質生命理工学科に異動した。大学では，白衣を着て何とあの南部が実験の授業をやっている。ある意味，スーパーコンピュータから離れた立場となったが，実験研究者が実験装置に工夫をするように，理論研究者がコンピュータを意識してプログラムを開発することは，とても大切なことだと考えている。その一方，ここに至るまで様々な方々にお世話になった。この場をお借りして感謝申し上げる。ありがとうございました。

平成 22 年 3 月末　南部伸孝

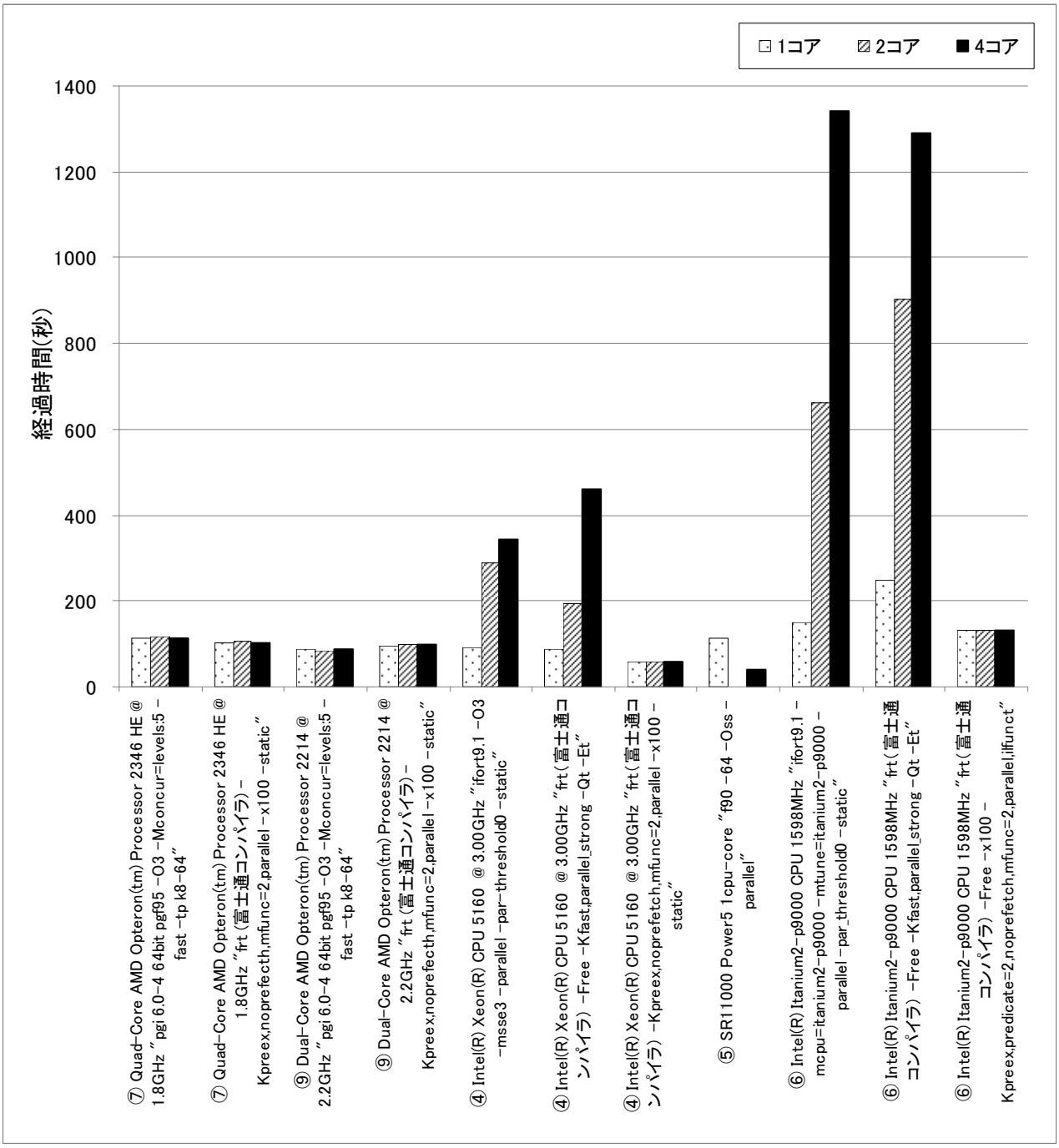


図 6. 様々なマシンと各社がリリースするコンパイラの自動並列化性能

【補足資料】

(1) 測定に用いたコード（4 次のシンプレクティク積分法）

```
! We use 4th order Gray sypmlectic integrator [JCP v106 p4832 (1997)]

! Coordinate integration coefficients
a(1) = 0.5*(1.0 - 0.5773502691896257645091487805019 6)*m_traj%tstep
a(2) = 0.5773502691896257645091487805019 6*m_traj%tstep
a(3) = -0.5773502691896257645091487805019 6*m_traj%tstep
a(4) = 0.5*1.5773502691896257645091487805019 6*m_traj%tstep

! Momentum integration coefficients
b(1) = 0.0
b(2) = 0.5*1.0773502691896257645091487805019 6*m_traj%tstep
b(3) = 0.5*m_traj%tstep
b(4) = -0.5*0.0773502691896257645091487805019 6*m_traj%tstep

! Make forward in time step for clasical trajectory

! Trajectory, monodromy matrixes and action are calculated
! simultaneosly to use advantage of local data cash
time = m_traj%tstep*m_traj%it
      ActionStep = 0.0
do k = 1, 4

      ! Data at q_{k-1}
      if(b(k) /= 0.0) then

            call tr_CashRightHandSides(m_traj, time)

            ! v_p = p_{k-1} -> p_{k}
            do i = 1, NUM_OF_DOF
                  m_traj%v_p(i) = m_traj%v_p(i) + b(k)*m_traj%c_force(i)
            enddo

            ! From now v_q = q_{k-1}, v_p = p_{k}, time = t_{k-1}
            do i = 1, NUM_OF_DOF
                  do j = 1, NUM_OF_DOF
                        m_dtmp(i, j) = 0.0
                        do m = 1, NUM_OF_DOF
                              m_dtmp(i, j) = m_dtmp(i, j) &
                                    & - m_traj%c_hess(i, m)*m_traj%m_QP(m, j)
                        enddo
                  enddo
            enddo
            do i = 1, NUM_OF_DOF
                  do j = 1, NUM_OF_DOF
                        m_traj%m_PP(i, j) = m_traj%m_PP(i, j) + b(k)*m_dtmp(i, j)
                  enddo
            enddo

            do i = 1, NUM_OF_DOF
                  do j = 1, NUM_OF_DOF
                        m_dtmp(i, j) = 0.0
                        do m = 1, NUM_OF_DOF
                              m_dtmp(i, j) = m_dtmp(i, j) &
                                    & - m_traj%c_hess(i, m)*m_traj%m_QQ(m, j)
                        enddo
                  enddo
            enddo
            do i = 1, NUM_OF_DOF
                  do j = 1, NUM_OF_DOF
                        m_traj%m_PQ(i, j) = m_traj%m_PQ(i, j) + b(k)*m_dtmp(i, j)
                  enddo
            enddo

      endif

      ! Calculate action step
      dtmp = 0.0        ! Get kinetic energy
      do i = 1, NUM_OF_DOF
            do j = 1, NUM_OF_DOF
                  dtmp = dtmp + m_traj%v_p(i)*m_traj%v_p(j) &
```

```
                    & *pes_GetKineticMatrix(i, j)
            enddo
        enddo
        if(b(k) /= 0.0) then
            ActionStep = ActionStep + a(k)*dtmp - b(k)*m_traj%c_pot
        else
            ActionStep = ActionStep + a(k)*dtmp
        endif

        ! v_q = q_{k-1} -> q_{k}
        do i = 1, NUM_OF_DOF
            v_dtmp(i) = 0.0
            do j = 1, NUM_OF_DOF
                if(i == j) then
                    v_dtmp(i) = v_dtmp(i) &
                        & + 2.0*m_traj%v_p(i)*pes_GetKineticMatrix(i, i)
                else
                    v_dtmp(i) = v_dtmp(i) &
                        & + m_traj%v_p(j)*pes_GetKineticMatrix(i, j)
                endif
            enddo
        enddo
        do i = 1, NUM_OF_DOF
            m_traj%v_q(i) = m_traj%v_q(i) + a(k)*v_dtmp(i)
        enddo

        ! From now v_q = q_{k}, v_p = p_{k}, time = t_{k-1}
        do i = 1, NUM_OF_DOF
            do j = 1, NUM_OF_DOF
                m_dtmp(i, j) = 0.0
                do m = 1, NUM_OF_DOF
                    if(i == m) then
                        m_dtmp(i, j) = m_dtmp(i, j) &
                            & + 2.0*pes_GetKineticMatrix(i, m)*m_traj%m_PP(m, j)
                    else
                        m_dtmp(i, j) = m_dtmp(i, j) &
                            & + pes_GetKineticMatrix(i, m)*m_traj%m_PP(m, j)
                    endif
                enddo
            enddo
        enddo
        do i = 1, NUM_OF_DOF
            do j = 1, NUM_OF_DOF
                m_traj%m_QP(i, j) = m_traj%m_QP(i, j) + a(k)*m_dtmp(i, j)
            enddo
        enddo

        do i = 1, NUM_OF_DOF
            do j = 1, NUM_OF_DOF
                m_dtmp(i, j) = 0.0
                do m = 1, NUM_OF_DOF
                    if(i == m) then
                        m_dtmp(i, j) = m_dtmp(i, j) &
                            & + 2.0*pes_GetKineticMatrix(i, m)*m_traj%m_PQ(m, j)
                    else
                        m_dtmp(i, j) = m_dtmp(i, j) &
                            & + pes_GetKineticMatrix(i, m)*m_traj%m_PQ(m, j)
                    endif
                enddo
            enddo
        enddo
        do i = 1, NUM_OF_DOF
            do j = 1, NUM_OF_DOF
                m_traj%m_QQ(i, j) = m_traj%m_QQ(i, j) + a(k)*m_dtmp(i, j)
            enddo
        enddo

        ! Time update time = t_{k}
        time = time + a(k)
        if(k == 4) time = m_traj%tstep*(m_traj%it + 1)
    enddo
```

(2) 詳細なコンパイラオプション及び計測経過時間

機種	分子動力学プログラム（自作）	経過時間（秒）
①	AMD Athlon(tm) 64 Processor 3500+ "ifort9.1-O3 -msse2"	159.029
①	AMD Athlon(tm) 64 Processor 3500+ "pgi 6.0-4 64bit pgf -O3 -fast -tp k8-64"	89.921
②	AMD Athlon(tm)64 X2 Dual Core Processor 4400+ "ifort9.1-O3 -msse2"	181.043
②	AMD Athlon(tm)64 X2 Dual Core Processor 4400+ "ifort9.1-O3 -msse2 -parallel -par_threshold0" & "setenv OMP_NUM_THREADS 2"	237.762
②	AMD Athlon(tm)64 X2 Dual Core Processor 4400+ "pgi 6.0-4 64bit pgf95 -O3 -fast -tp k8-64"	84.761
②	AMD Athlon(tm)64 X2 Dual Core Processor 4400+ "pgi 6.0-4 64bit pgf95 -O3 -Mconcur=levels:5 -fast -tp k8-64" & "setenv NCPUS 1"	83.529
②	AMD Athlon(tm)64 X2 Dual Core Processor 4400+ "pgi 6.0-4 64bit pgf95 -O3 -Mconcur=levels:5 -fast -tp k8-64" & "setenv NCPUS 2"	88.225
②	AMD Athlon(tm)64 X2 Dual Core Processor 4400+ "frt（富士通コンパイラ）-Kpreex,noprefecth,mfunc=2,parallel -x100 -static" & "setenv OMP_NUM_THREADS 1"	96.620
②	AMD Athlon(tm)64 X2 Dual Core Processor 4400+ "frt（富士通コンパイラ）-Kpreex,noprefecth,mfunc=2,parallel -x100 -static" & "setenv OMP_NUM_THREADS 2"	97.170
⑦	Quad-Core AMD Opteron(tm) Processor 2346 HE @ 1.8GHz "pgi 6.0-4 64bit pgf95 -O3 -Mconcur=levels:5 -fast -tp k8-64" & "setenv NCPUS 1"	114.180
⑦	Quad-Core AMD Opteron(tm) Processor 2346 HE @ 1.8GHz "pgi 6.0-4 64bit pgf95 -O3 -Mconcur=levels:5 -fast -tp k8-64" & "setenv NCPUS 2"	115.700
⑦	Quad-Core AMD Opteron(tm) Processor 2346 HE @ 1.8GHz "pgi 6.0-4 64bit pgf95 -O3 -Mconcur=levels:5 -fast -tp k8-64" & "setenv NCPUS 4"	114.710
⑦	Quad-Core AMD Opteron(tm) Processor 2346 HE @ 1.8GHz "frt（富士通コンパイラ）-Kpreex,noprefecth,mfunc=2,parallel -x100 -static" & "setenv OMP_NUM_THREADS 1"	102.300
⑦	Quad-Core AMD Opteron(tm) Processor 2346 HE @ 1.8GHz "frt（富士通コンパイラ）-Kpreex,noprefecth,mfunc=2,parallel -x100 -static" & "setenv OMP_NUM_THREADS 2"	104.310
⑦	Quad-Core AMD Opteron(tm) Processor 2346 HE @ 1.8GHz "frt（富士通コンパイラ）-Kpreex,noprefecth,mfunc=2,parallel -x100 -static" & "setenv OMP_NUM_THREADS 4"	102.210
⑨	Dual-Core AMD Opteron(tm) Processor 2214 @ 2.2GHz "pgi 6.0-4 64bit pgf95 -O3 -Mconcur=levels:5 -fast -tp k8-64" & "setenv NCPUS 1"	86.530
⑨	Dual-Core AMD Opteron(tm) Processor 2214 @ 2.2GHz "pgi 6.0-4 64bit pgf95 -O3 -Mconcur=levels:5 -fast -tp k8-64" & "setenv NCPUS 2"	85.690
⑨	Dual-Core AMD Opteron(tm) Processor 2214 @ 2.2GHz "pgi 6.0-4 64bit pgf95 -O3 -Mconcur=levels:5 -fast -tp k8-64" & "setenv NCPUS 4"	86.510
⑨	Dual-Core AMD Opteron(tm) Processor 2214 @ 2.2GHz "frt（富士通コンパイラ）-Kpreex,noprefecth,mfunc=2,parallel -x100 -static" & "setenv OMP_NUM_THREADS 1"	96.180
⑨	Dual-Core AMD Opteron(tm) Processor 2214 @ 2.2GHz "frt（富士通コンパイラ）-Kpreex,noprefecth,mfunc=2,parallel -x100 -static" & "setenv OMP_NUM_THREADS 2"	99.120
⑨	Dual-Core AMD Opteron(tm) Processor 2214 @ 2.2GHz "frt（富士通コンパイラ）-Kpreex,noprefecth,mfunc=2,parallel -x100 -static" & "setenv OMP_NUM_THREADS 4"	98.290
③	Intel(R) Pentium(R) 4 CPU 3.40GHz "ifort9.1 -O3 -msse2"	88.807
③	Intel(R) Pentium(R) 4 CPU 3.40GHz "ifort9.1 -O3 -msse3"	78.825
④	Intel(R) Xeon(R) CPU 5160 @ 3.00GHz "ifort9.1 -O3 -msse3 -parallel -par-threshold0 -static" & "setenv OMP_NUM_THREADS 1"	89.800
④	Intel(R) Xeon(R) CPU 5160 @ 3.00GHz "ifort9.1 -O3 -msse3 -parallel -par-threshold0 -static" & "setenv OMP_NUM_THREADS 2"	287.730
④	Intel(R) Xeon(R) CPU 5160 @ 3.00GHz "ifort9.1 -O3 -msse3 -parallel -par-threshold0 -static" & "setenv OMP_NUM_THREADS 4"	343.050
④	Intel(R) Xeon(R) CPU 5160 @ 3.00GHz "frt（富士通コンパイラ） -Free -Kfast -Qt -Et"	75.232
④	Intel(R) Xeon(R) CPU 5160 @ 3.00GHz "frt（富士通コンパイラ） -Free -Kfast,parallel_strong -Qt -Et" & "setenv OMP_NUM_THREADS 1"	86.670
④	Intel(R) Xeon(R) CPU 5160 @ 3.00GHz "frt（富士通コンパイラ） -Free -Kfast,parallel_strong -Qt -Et" & "setenv OMP_NUM_THREADS 2"	194.640
④	Intel(R) Xeon(R) CPU 5160 @ 3.00GHz "frt（富士通コンパイラ） -Free -Kfast,parallel_strong -Qt -Et" & "setenv OMP_NUM_THREADS 4"	461.120

④	Intel(R) Xeon(R) CPU 5160 @ 3.00GHz "frt（富士通コンパイラ） -Kpreex,noprefetch,mfunc=2, parallel -x100 -static" & "setenv OMP_NUM_THREADS 1"	58.590
④	Intel(R) Xeon(R) CPU 5160 @ 3.00GHz "frt（富士通コンパイラ） -Kpreex,noprefetch,mfunc=2, parallel -x100 -static" & "setenv OMP_NUM_THREADS 2"	58.640
④	Intel(R) Xeon(R) CPU 5160 @ 3.00GHz "frt（富士通コンパイラ） -Kpreex,noprefetch,mfunc=2, parallel -x100 -static" & "setenv OMP_NUM_THREADS 4"	58.730
⑧	Intel(R) Core™2 Duo CPU E6850 @ 3.00GHz "ifort9.1 -O3 -msse3 -parallel -par-threshold0 -static" & "setenv OMP_NUM_THREADS 1"	77.900
⑧	Intel(R) Core™2 Duo CPU E6850 @ 3.00GHz "ifort9.1 -O3 -msse3 -parallel -par-threshold0 -static" & "setenv OMP_NUM_THREADS 2"	101.600
⑧	Intel(R) Core™2 Duo CPU E6850 @ 3.00GHz "frt（富士通コンパイラ） -Kpreex,noprefetch,mfunc=2,parallel -x100 -static" & "setenv OMP_NUM_THREADS 1"	59.980
⑧	Intel(R) Core™2 Duo CPU E6850 @ 3.00GHz "frt（富士通コンパイラ） -Kpreex,noprefetch,mfunc=2,parallel -x100 -static" & "setenv OMP_NUM_THREADS 2"	58.600
⑤	SR11000 Power5 1cpu-core "f90 -64 -Oss -parallel" & "setenv HF_PRUNST_THREADNUM 1"	113.329
⑤	SR11000 Power5 1cpu-core "f90 -64 -Oss -parallel" & "setenv HF_PRUNST_THREADNUM 4"	38.550
⑥	Intel(R) Itanium2-p9000 CPU 1598MHz "ifort9.1 -mcpu=itanium2-p9000 -mtune=itanium2-p9000 -static"	49.978
⑥	Intel(R) Itanium2-p9000 CPU 1598MHz "ifort9.1 -mcpu=itanium2-p9000 -mtune=itanium2-p9000 -parallel -par_threshold0 -static" & "setenv OMP_NUM_THREADS 1"	148.610
⑥	Intel(R) Itanium2-p9000 CPU 1598MHz "ifort9.1 -mcpu=itanium2-p9000 -mtune=itanium2-p9000 -parallel -par_threshold0 -static" & "setenv OMP_NUM_THREADS 2"	661.780
⑥	Intel(R) Itanium2-p9000 CPU 1598MHz "ifort9.1 -mcpu=itanium2-p9000 -mtune=itanium2-p9000 -parallel -par_threshold0 -static" & "setenv OMP_NUM_THREADS 4"	1340.150
⑥	Intel(R) Itanium2-p9000 CPU 1598MHz "frt（富士通コンパイラ） -Free -Kfast -Qt -Et"	170.916
⑥	Intel(R) Itanium2-p9000 CPU 1598MHz "frt（富士通コンパイラ） -Free -Kfast,parallel_strong -Qt -Et" & "setenv OMP_NUM_THREADS 1"	248.665
⑥	Intel(R) Itanium2-p9000 CPU 1598MHz "frt（富士通コンパイラ） -Free -Kfast,parallel_strong -Qt -Et" & "setenv OMP_NUM_THREADS 2"	902.400
⑥	Intel(R) Itanium2-p9000 CPU 1598MHz "frt（富士通コンパイラ） -Free -Kfast,parallel_strong -Qt -Et" & "setenv OMP_NUM_THREADS 4"	1290.110
⑥	Intel(R) Itanium2-p9000 CPU 1598MHz "frt（富士通コンパイラ） -Free -x100 -Kpreex,predicate=2,noprefetch,mfunc=2,ilfunct"	150.560
⑥	Intel(R) Itanium2-p9000 CPU 1598MHz "frt（富士通コンパイラ） -Free -x100 -Kpreex,predicate=2,noprefetch,mfunc=2,parallel,ilfunct" & "setenv OMP_NUM_THREAD 1"	131.770
⑥	Intel(R) Itanium2-p9000 CPU 1598MHz "frt（富士通コンパイラ） -Free -x100 -Kpreex,predicate=2,noprefetch,mfunc=2,parallel,ilfunct" & "setenv OMP_NUM_THREAD 2"	132.110
⑥	Intel(R) Itanium2-p9000 CPU 1598MHz "frt（富士通コンパイラ） -Free -x100 -Kpreex,predicate=2,noprefetch,mfunc=2,parallel,ilfunct" & "setenv OMP_NUM_THREAD 4"	132.350

2.3.4. LINPACK および OS ジッタについて

富士通株式会社　　青木 正樹

ここでは、FX1 での LINPACK 実行性能および OS ジッタの影響を検証したので報告する。

1. FX1 の LINPACK 自由元性能

(1) FX1 性能

ノード数	搭載メモリ (GB)	使用コア数	Rpeak (TFLOPS)	Rmax (TFLOPS)	効率
1	32	4	0.040	0.03702	91.82%
384	16	1536	15.48	13.61	87.90%
512	32	2048	20.64	18.54	89.81%

(2) 実行効率（2009.1.29 時点）

順位	システム	使用コア数	Rpeak (TFLOPS)	Rmax (TFLOPS)	効率
1	Altix4700　1.6GHz	9728	62.26	56.52	90.78%
2	FX1	2084	20.64	18.54	89.81%
3	Altix4700　1.6GHz	2560	16.38	14.59	89.07%
4	AltixICE Xeon quad	2560	28.67	25.11	87.58%
5	Earth-Simulator	5120	40.96	35.86	87.55%

FX1 以外のデータは、TOP500 2008.11 リストより。TOP500 平均は、62.64%。

(3) CPU アーキ別 実行効率比較（2009.1.29 時点）

順位	システム	効率	備考
1	IPF	90.78%	SGI Altix
2	SPARC64VII	89.81%	FX1
3	Intel Xeon Quad core	87.58%	SGI Altix
4	ベクトル	87.55%	Earth-Simulator
5	AMD Opteron Dual core	86.15%	CRAY XT4
6	POWER5+	85.96%	IBM
7	POWERPC 450	84.08%	BlueGene/P
8	AMD Opteron Quad core	83.11%	T2K スパコン
9	POWER6	81.28%	IBM
10	PowerXCell	74.58%	IBM ペタコン

(4) FX1 上での高速化技術

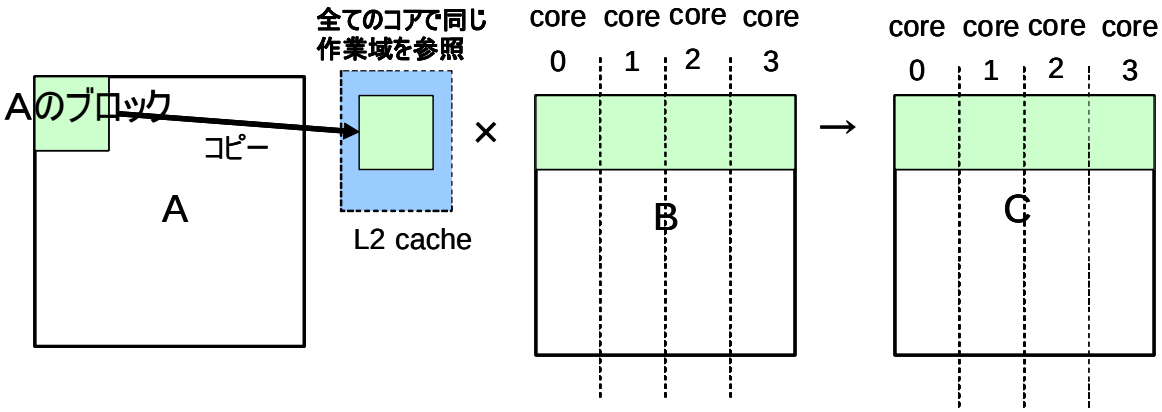
① DGEMM 高速化技術

ハードウェアのピーク性能の 94.6%という高性能を達成

■ 1 コア向けのチューニング
- 演算器の特性を活かした命令スケジューリング
- キャッシュ構成を想定したデータ配置
- 適切なプリフェッチ命令の挿入

■　マルチコア向けのチューニング

・　各コアが参照する配列を共有 L2 キャッシュ上で共有できるように配置することで、キャッシュミスを大幅に低減



② 大規模問題の実行

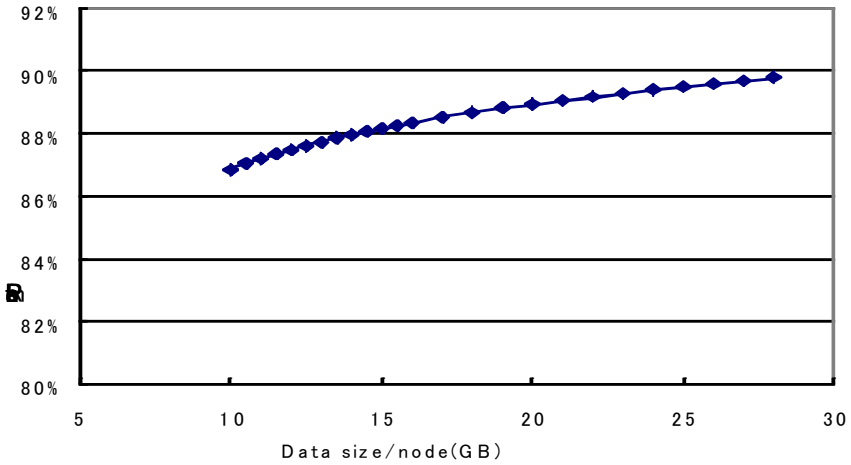
■ Linpack は問題規模を大きくすることで性能向上が可能

■ 今回の測定

・ 問題規模をメモリに格納できる最大規模(NMax：3,308,800)とし、実行時間を制約せずに測定を実施

・ 約 60 時間に及ぶ長時間を平均 91.19%の性能効率で完走することに成功

➔ 問題規模 1/2 で実行時間を短縮した場合と比べて約 2%の性能向上実現



2. OS ジッタについて

OS のノイズ（各種デーモンやタイマ割り込み等）が大規模並列アプリの実行性能に影響に大きな影響を与えることは知られている。

■　どのようなアプリが OS ジッタの影響を受けるのか？

⇒同期処理(バリアやリダクション)処理を頻繁に行うアプリが OS ジッタの影響を受けやすい。（通常 LINPACK は、同期処理が無い（少ない？）ために OS ジッタの影響は少ないと言われている。）

■　OS ジッタ対策は？

⇒一般的に以下がある。

① SMP ならば、1 つのコアを OS 専用に割り付ける（HPC2500 の用途分割がこれに対応）

② OS デーモンを軽くする
③ 協調スケジューリング（一時期にデーモン処理を集中実行させる）

FX1 では、①（SMT で実行）、③で対応。

3. LINPACK の OS ジッタ影響は？

FX1 では、SPARC64VII の特徴（共用キャッシュ）を活かし、チップ内はスレッド並列で実行している。LINPACK の核である DGEMM 実行時の OS ノイズ（デーモン）の影響を検証し、微小ではあるが影響があることを確認した。

■ 検証方法

以下の 2 通りの環境で 1 チップ内のスレッド並列化された DGEMM を実行し、影響を確認する。
・通常の運用モードでの実行
・デーモン抑止モードでの実行

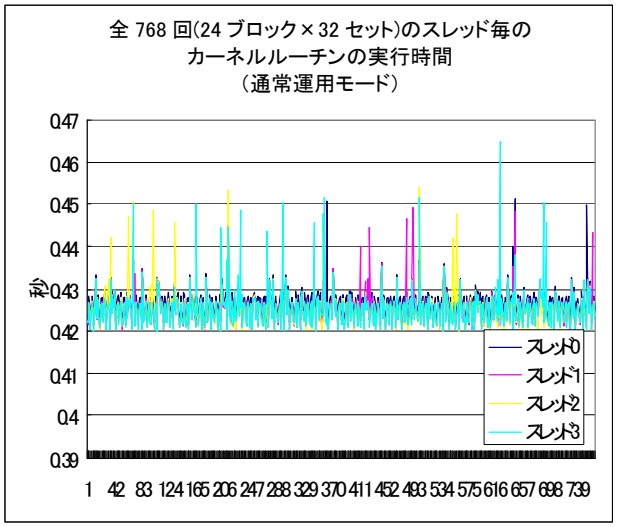
■ 結果

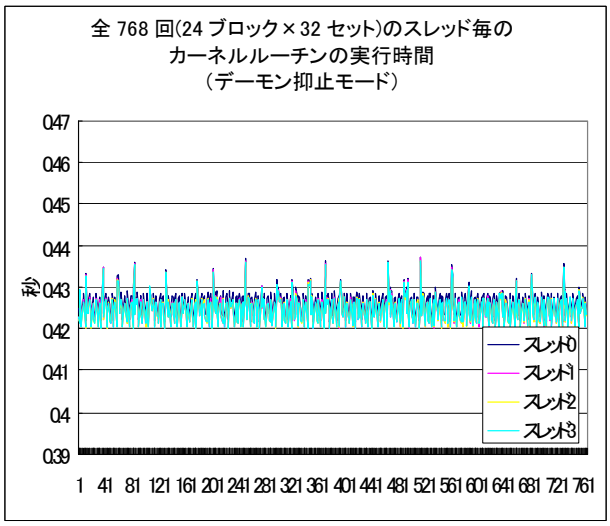
少なくても、0.24%の性能影響を確認。
注) 大規模ノードでの LINPACK 全体への影響は、実行環境面で測定困難であるため、DGEMM レベルの検証に留めた。

■ OS ジッタ影響での性能値

- 通常運用モード　：38.239 GFLOPS　ピーク比 94.84%
- デーモン抑止モード：38.335 GFLOPS　ピーク比 95.08%

差 0.24 %





以上

[添付]

PAイベント情報活用チュートリアル -サイクルアカウンティング　FX1編-

富士通株式会社　青木正樹

目次

- PAイベント情報概略
- 代表的なPAイベント情報
  - 演算コミット数
  - メモリアクセス待ち
  - 浮動少数点演算待ち
  - キャッシュアクセス待ち
- チューニング事例
- PAイベント情報採取ツール

サイクルアカウンティングとは

性能ボトルネック要因分析の手法である。SPARC64VIIでは豊富なPAイベント情報を備えており、アプリケーションプログラム実行時のCPU動作状態の情報が取得できる。あるアプリケーションプログラムを実行するためにかかった総時間（CPUサイクル数）をCPUの動作状態で分類(*)し、CPU内のどの部分にボトルネックがあるかを把握することで、詳細な性能分析やチューニングを行うことができる。

(*)命令実行中、メモリアクセス待ち、演算完了待ちである、など

予告：次世代スパコン向けにも＋α充実させていきます。目標2010年末

PAイベント情報概略

最大4コミット	完了命令数が4である時間
整数レジスタ書き込み制約	整数レジスタを2個更新しているため、4命令完了できなかった時間、命令完了数は2もしくは3
その他コミット	整数レジスタ書き込み制約を伴わない、完了命令数が1，2，3のいずれかである時間
メモリアクセス待ち	主記憶からのデータ待ちのため命令完了数が0である時間
キャッシュアクセス待ち	キャッシュからのデータ待ちのため命令完了数が0である時間
浮動小数点演算待ち	実行中の命令の中で一番古い命令が、浮動小数点の演算実行中で、命令完了数が0である時間
ストア待ち	ストアポートフル(*1)要因でCSE(*2)が空となっている時間
命令フェッチ待ち(*3)	ストアポートフル要因以外でCSEが空となっている時間
その他待ち	メモリアクセス待ち、キャッシュアクセス待ち、浮動小数点演算待ち、ストア待ち、命令フェッチ待ち以外の要因で、命令完了数が0である時間

(*1) ストアポートフル
ストアポートとはストアデータを格納するキューです。SPARC64™ VIIIfxはストアポートが満杯となった時点でストアポートに空きができるまで、以降の命令デコードを停止します。ストアポートフルの典型的なトリガはストア命令のデータキャッシュミスです。

(*2) CSE(コミットスタックエントリ)
CSEとは実行中の（発行されてまだ完了していない）命令の情報を保持するためのバッファです。

(*3) 命令フェッチ待ち
典型的なケースは以下の2つです。
- 命令キャッシュミスによりCPU内で実行すべき命令を供給できない
- スレッド並列実行において、ソフトから他コアとの同期を指示され同期成立を待っている

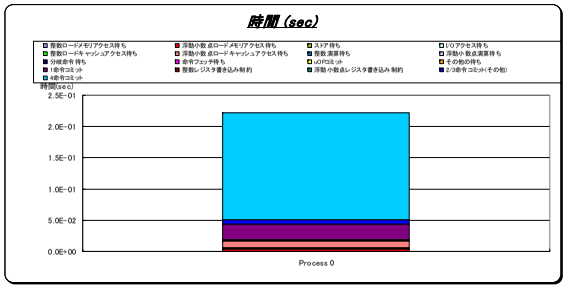
代表的なPAイベント情報

ーどのようなコード？ どのような情報？ー

代表的なPA情報：演算コミット数

プログラム例：行列積

```
A=MATMUL(B,C)
```



0 Commit
1 Commit
2-3 Commit
4 Commit

1マシンサイクルでn命令実行したことを示す。

OCommit部分は、何らかの要因で命令が動作していない時間。

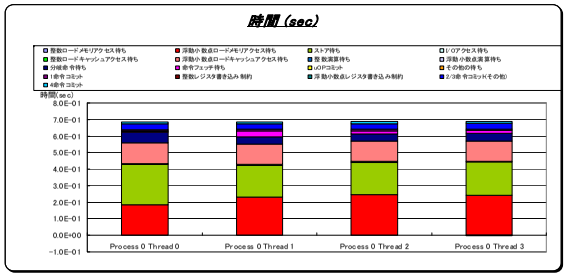
※モノクロ印刷ではグラフの色識別が不鮮明です。SS研Webサイト「WG成果報告書」のPDFでカラー版をご覧ください。 以降同様。

代表的なPA情報：メモリアクセス待ち

プログラム例：STREAM COPY

```
DO I=1,L2$に乗らない大きさ
  A(I)=B(I)
ENDDO
```

浮動小数点ロードメモリアクセス待ち
整数ロードメモリアクセス待ち
ストア待ち

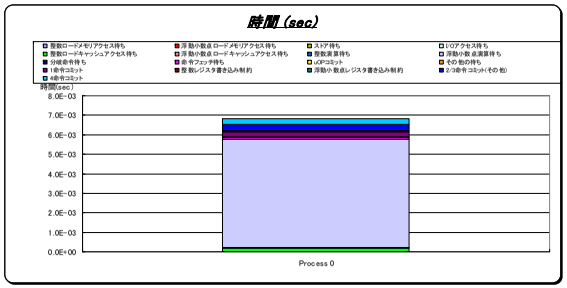


浮動小数点ロードメモリアクセス待ち、整数ロードメモリアクセス待ち、ストア待ち部分は、それぞれのメモリアクセス命令が引き起こすL2$ミス要因で他の命令が動作していない時間。

代表的なPA情報：浮動少数点演算待ち

プログラム例：除算

```
DO I=1,M    配列A,B,CはL1$上
  A(I)=B(I)/C(I)
ENDDO
```



浮動小数点演算待ち

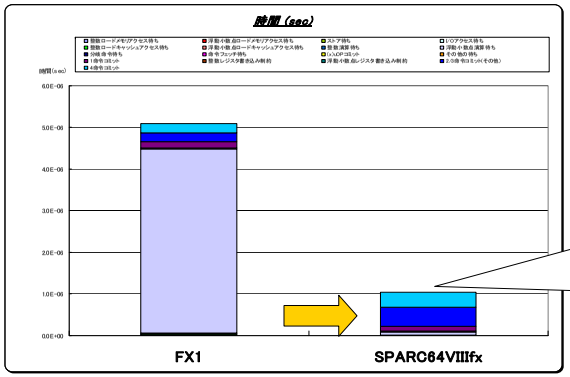
浮動小数点演算待ち部分は、浮動小数点演算命令が要因で他の命令が動作していない時間。

除算はレイテンシの長い、かつ、パイプライン実行できないハード命令で実装されているため、浮動小数点演算待ちが発生。

参考）ただしSPARC64Viiifxでの除算は

プログラム例：除算

```
DO I=1,M    配列A,B,CはL1$上
  A(I)=B(I)/C(I)
ENDDO
```

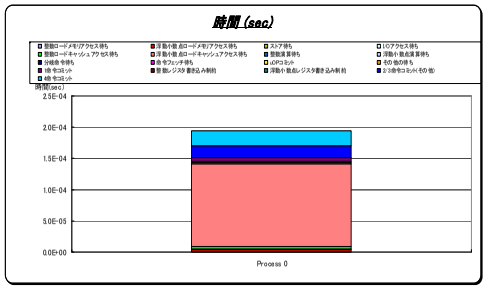


FX1性能比
約5倍向上
除算影響の待ちは
無くなる

代表的なPA情報：キャッシュアクセス待ち

プログラム例

```
DO J=1,M
DO I=1,N    配列A,BはL2$上
  A(I,J)=B(I,J)+1.0
ENDDO
ENDDO
```



浮動小数点ロードキャッシュアクセス待ち

整数ロードキャッシュアクセス待ち

浮動小数点ロードキャッシュアクセス待ち、整数ロードキャッシュアクセス待ちは、それぞれのメモリアクセス命令がキャッシュアクセスの待ち要因で他の命令が動作していない時間。

チューニング事例

- 1次キャッシュ・スラッシング回避
- リストアクセスに対するプリフェッチ
- キャッシュ・チューニング（多重ループ融合）
- スレッド並列のロードバランス不均等
- リストアクセスのチューニング（配列マージ）
- メモリアクセスの局所化（タイリング）

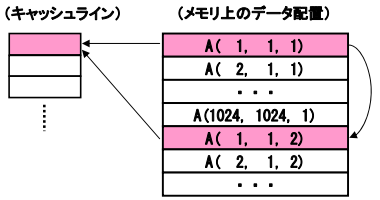
1次キャッシュ・スラッシング回避（前）

SPARC64ViiのL1D$キャッシュは、64KB/2Wayです。

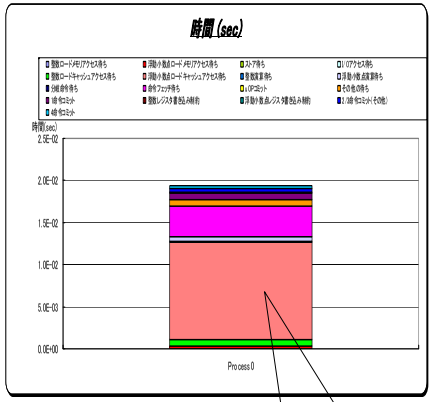
配列の形状が2のべき乗になっていると、キャッシュ競合で性能低下する場合があります。

オリジナルソース

```
REAL*8 A(1024, 1024, 4)
DO I=1,N
  A(I,1,1)=A(I,1,2)+A(I,1,3)+A(I,1,4)
ENDDO
```



同じキャッシュラインへの競合



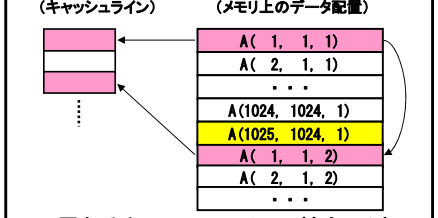
キャッシュ待ち時間

1次キャッシュ・スラッシング回避（後）

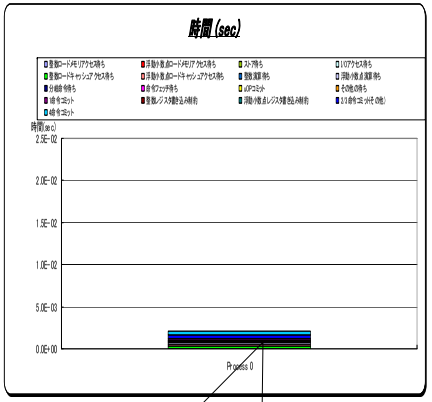
パディング（内側にすき間を空ける）を行うことにより、キャッシュ競合を削減します。

コンパイラ最適化のイメージ

```
REAL*8 A(1025, 1024, 4)
DO I=1,N
  A(I,1,1)=A(I,1,2)+A(I,1,3)+A(I,1,4)
ENDDO
```



異なるキャッシュラインで競合回避



キャッシュ待ち時間激減

リストアクセスに対するプリフェッチ（前）

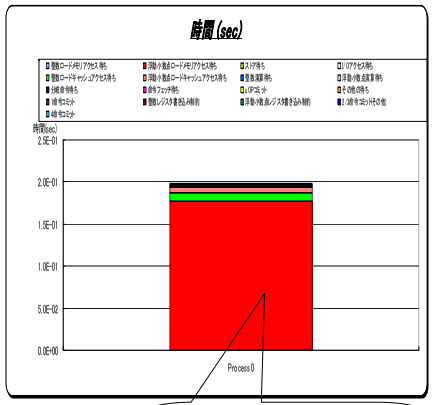
配列のリスト参照のようなインダイレクト（リスト）アクセスについては、-Kfastオプションでは、プリフェッチ命令を生成しません。

オリジナルソース

```
REAL*8 A(N),B(N),C(N)
REAL*8 D(N),E(N),F(N)
DO I=1,N
```

配列D,E,Fに対して
プリフェッチ命令を生成

```
A(D(I))=B(E(I))+SCALAR*C(F(I))
ENDDO
```



リストアクセスのメモリアクセス待ち時間

リストアクセスに対するプリフェッチ（後）

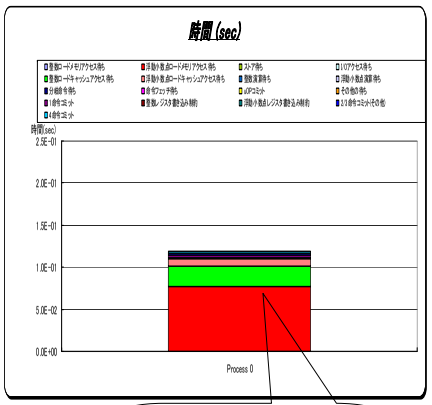
コンパイラオプション-Kprefetch_indirectもしくは最適化制御行による直接指示でリストアクセスについてもプリフェッチ命令を生成することができます。

コンパイラ最適化のイメージ

```
REAL*8 A(N),B(N),C(N)
REAL*8 D(N),E(N),F(N)
DO I=1,N
```

配列A,B,C,D,E,Fに対して
プリフェッチ命令を生成

```
A(D(I))=B(E(I))+SCALAR*C(F(I))
ENDDO
```



プリフェッチにより
待ち時間減少

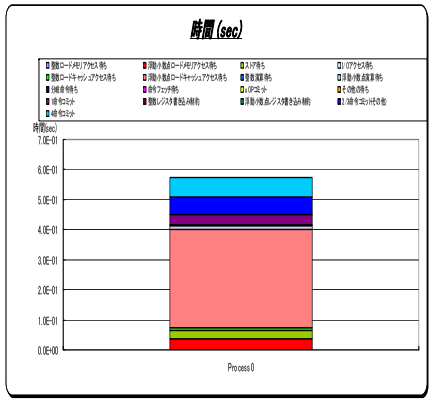
キャッシュ・チューニング：多重ループ融合（前）

キャッシュのチューニングとは、時間的/空間的に近くをアクセスすることです。チューニングの際は、メモリアクセスの状況を大域的に見てチューニングすることを薦めます。

オリジナルソース

```
REAL*8 A(N,N),B(N,N),C(N,N)
DO J=1,N
  DO I=1,N
    A(I,J)=B(I,J)+C(I,J)
  ENDDO
ENDDO
DO J=1,N-1
  DO I=1,N
    C(I,J)=A(I,J)+B(I,J)+1.0
  ENDDO
ENDDO
```

N=1000



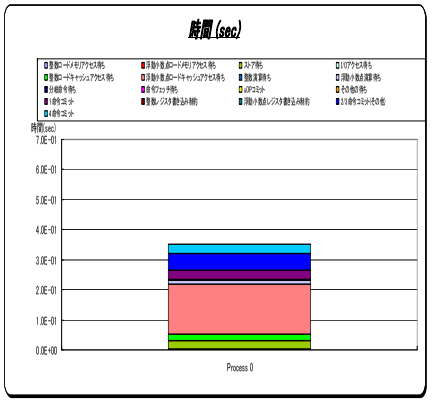
配列のアクセス時間が全体の2/3

キャッシュ・チューニング：多重ループ融合（後）

LOOP-PEELINGを行い、2つの2重ループを融合することにより、キャッシュアクセスのチューニングを行った例です。

チューニング後ソース

```
REAL*8 A(N,N),B(N,N),C(N,N)
A(N,N)=B(N,N)+C(N,N)
DO J=1,N-1
  DO I=1,N
    A(I,J)=B(I,J)+C(I,J)
    C(I,J)=A(I,J)+B(I,J)+1.0
  ENDDO
ENDDO
J=N
DO I=1,N
  A(I,J)=B(I,J)+C(I,J)
ENDDO
```



配列のアクセス時間が半減

スレッド並列のロードバランス不均等（前）

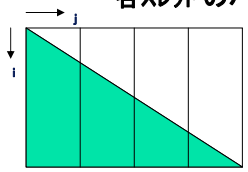
並列チューニングの観点から、解決しなければならない課題にロードバランスの均等化があります。スレッド並列の三角ループを外側ループで均等に割り付ける場合は、注意が必要です。

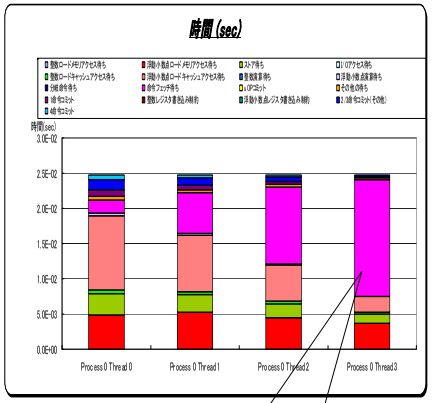
オリジナルソース

```
COMMON A,B,C,D
REAL*8 A(4097,4096),B(4097,4096),C(4097,4096)
!$OMP PARALLEL DO
DO J=1,4096
  DO I=J,4096
    A(I,J)=B(I,J)+C(I,J)
  ENDDO
ENDDO
```

三角ループ

各ｽﾚｯﾄﾞのバランス悪い！





バリア待ち時間

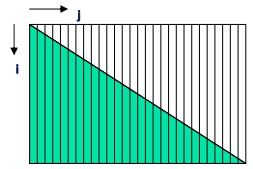
スレッド並列のロードバランス不均等（後）

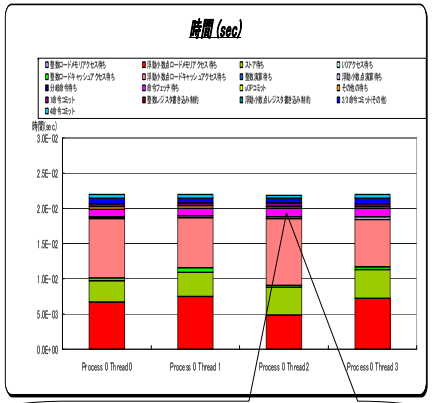
三角ループは、サイクリック割付を行うことにより、ロードバランスの均等化が可能となります。

チューニング後ソース

```
COMMON A,B,C,D
REAL*8 A(4097,4096),B(4097,4096),C(4097,4096)
!$OMP PARALLEL DO SCHEDULE(STATIC,1)
DO J=1,4096
  DO I=J,4096
    A(I,J)=B(I,J)+C(I,J)
  ENDDO
ENDDO
```

三角ループ





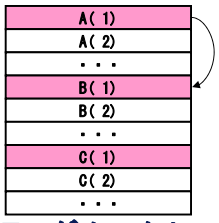
バリア待ち時間均一。ただし、1刻みだとL1$のFalse sharing影響見える

リストアクセスのチューニング：配列マージ（前）

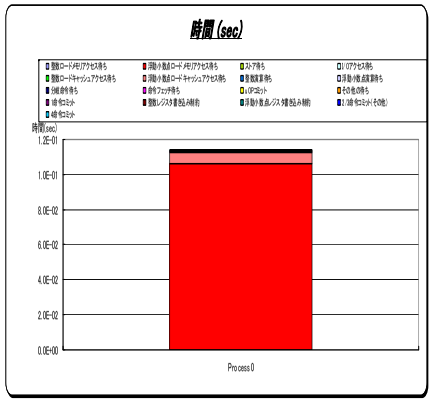
リストアクセス時、リストの値が不連続＆遠い場合、キャッシュの使用効率が悪い場合があります。

オリジナルソース

```
REAL*8 A(N),B(N),C(N)
DO I=1,N
  A(L(I))=B(L(I))+C(L(I))
ENDDO
```



ランダムアクセス



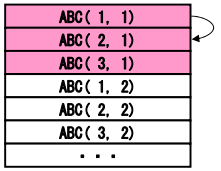
⇒キャッシュの使用効率が良くない

リストアクセスのチューニング：配列マージ（後）

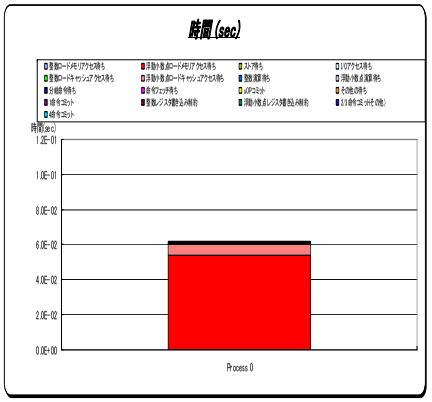
複数の配列を1個にマージすることでデータアクセスが改善される可能性があります。
最適化制御行！ocl array_mergeも用意されています。

融合後イメージ

```
REAL*8 ABC(3,N)
DO I=1,100
  ABC(1,L(I))=ABC(2,L(I))+ABC(3,L(I))
ENDDO
```



連続アクセス



⇒キャッシュ使用効率向上

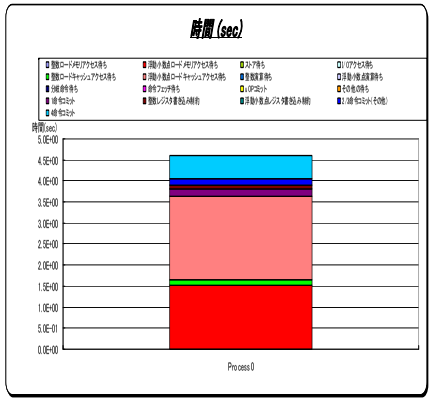
メモリアクセスの局所化：タイリング（前）

大きな配列の転置は、キャッシュの使用効率が良くありません。

オリジナルソース

```
REAL*8 A(1000,1000),B(1000,1000)
DO I=1,N
   DO J=1,N
      A(I,J)=B(J,I)
   ENDDO
ENDDO
```

配列Bは1000要素飛びアクセス



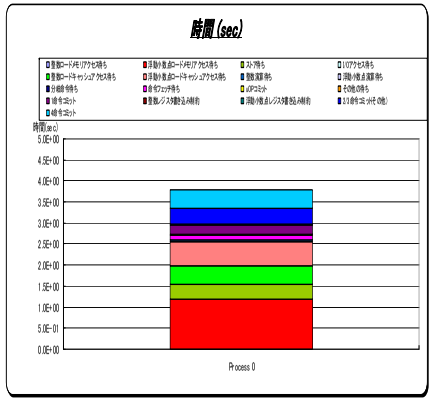
メモリアクセスの局所化：タイリング（後）

メモリアクセスを局所化するために,配列をM×Mサイズに分割します。これをタイル化といいます。(Mのサイズはアクセス状況 / キャッシュサイズに依存します)　今回は、タイルサイズ40での実測結果。

チューニング後ソース

```
REAL*8 A(1000,1000),B(1000,1000)
M=40
DO II=1,N,M
  DO JJ=1,N,M
    DO I=II,MIN(II+M,N)
      DO J=JJ=MIN(JJ+M,N)
        A(I,J)=B(J,I)
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

配列BはM要素飛びアクセス



3. プログラミングモデルサブ WG 報告

3.1. サブ WG まとめ

プログラミングモデルサブ WG まとめ役
山口大学大学院医学系研究科　　平野　靖
(前・名古屋大学情報基盤センター)

このサブ WG では、言語比較、MPI、フラット MPI とハイブリッド並列という 3 つの切り口で HPC の使い方の検討を行なった。

現在の HPC 環境では、Fortran だけでなく C 言語や C++などが使われる機会も増えてきた。また、Fortran90 では、FORTRAN77 では用いることができなかった構造体や動的配列確保、ポインタなどの仕様が追加され、より抽象度が高いプログラムや、物理モデルに即したデータ構造を持ったプログラムが作成可能になってきた。さらに、並列化の方法も自動並列化、OpenMP、あるいは MPI など様々なものが考えられる。

このように HPC におけるプログラム環境が多様化したことによって、ユーザの利便性は大幅に向上した。一方で、計算したい対象をどの言語でどのように構造化し、どのように並列化すれば良いかが不明確になった。

そこで、このサブ WG では、実コードあるいは実コードの一部を用いて、C 言語、C++および Fortran の性能比較・並列化時のラージページの取得方法の検討、構造体や module などの Fortran90 で導入された機能とその性能の検討、MPI の詳説と MPI-IO の検証、および並列プログラミングモデルの検証を行なった。このサブ WG の報告がユーザにとって、利便性と性能を両立させるプログラムの作成に役立つことを期待する。

3.2. 言語比較

3.2.1. Fortran と C 言語，C++の速度比較

山口大学大学院医学系研究科　　平野　靖
(前・名古屋大学情報基盤センター)

1. はじめに

スパコンやPCクラスタなどを用いた科学計算においては, Fortranが用いられることが多い. しかし, とくに大学の情報基盤センターなどではユーザ層の拡大のためにこれまで Fortran を使ってこなかった研究者の取り込みを行なう必要がある. Fortran の文法は他の高級言語と比較して単純であり制約が強い反面，他の言語と比較して最適化が簡単であるため，高速な実行プログラムを生成可能である．また，高速な実行プログラムを生成可能であることから科学計算を行なう研究者が好んで Fortran を用い，さらに利用者数が多いことからコンパイラ開発者がさらなる高速化を行なうという図式があると考えられる．一方，C 言語や C++などの言語は，Fortran と比較すると抽象化レベルが高いため直感的なプログラムを作成できるという長所があるが，文法に大きな自由度がある上に，ポインタ操作が多く用いられるため，あまり高度な最適化ができない．その結果として高速な実行プログラムが生成できない，という問題がある.

そこで，本文では，Fortran，C 言語および C++で同様の処理を行なう際の計算速度を評価する.

2. 測定に使用する処理とプログラムの概要

速度比較を行なう処理は，逐次処理による「行列同士の積」と「計算結果のファイルへの書き出し」とし，それぞれに要する CPU 時間を測定した．なお，行列のサイズは 1000×1000 とし，ファイルへの書き出しの際の通信の影響を避けるために, 計算結果は計算を行なった計算機のローカルディスク(/tmp)に書き出した.

また，言語，メモリの確保方法，およびファイルへの書き出し方法の違いによる性能の差異を比較するため, 下記のような同じ処理を行なう 24 個のプログラムを作成した. 表 1 にプログラムの概要を示す. また，作成したプログラムのうちのいくつかを文末の【補足資料】に例示する．表 1 で，例えばプログラム番号 17 は「2 次元配列」が×であり，「動的確保」と「構造体」,「書式付」が○となっている．これは，プログラム番号 17 では，構造体のメンバーとなっている行列を 1 次元的に動的確保し，行列積の計算結果を書式付きで書き出すことを意味する.

[C 言語]　8 個
- 行列を 1 次元配列で確保するか 2 次元配列で確保するか
- 配列(1 次元あるいは 2 次元)を静的に確保するか動的に確保するか
- 配列を構造体のメンバーにするか否か

[C++]　4 個
- 行列を 1 次元配列で確保するか 2 次元配列で確保するか
- 配列(1 次元あるいは 2 次元)を静的に確保するか動的に確保するか

なお，C++においては，すべてのプログラムで配列をクラスのメンバーとした.

[Fortran]　12 個
- 行列を 1 次元配列で確保するか 2 次元配列で確保するか
- 配列(1 次元あるいは 2 次元)を静的に確保するか動的に確保するか
- 配列を構造体のメンバーにするか否か
- 配列をファイルに書き出す際の書式の有無

なお，行列を 1 次元配列あるいは 2 次元配列で確保したときに，静的確保＆構造体＆書式無および動的確保＆構造体＆書式無のプログラムは作成していない．この理由として，たとえば静的確保＆構造体＆書式無の行列積に要する時間は静的確保＆構造体＆書式付と同様であり，書式の有無によるファイル

書き出しに要する時間の違いは静的確保＆非構造体＆書式付と静的確保＆非構造体＆書式無を比較することで推測可能であるからである．

表 1. 測定に使用したプログラムの条件

	C 言語								C++				Fortran											
プログラム番号	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
2 次元配列	×	×	×	×	○	○	○	○	×	×	○	○	×	×	×	×	×	×	○	○	○	○	○	○
動的確保	○	×	○	×	○	×	○	×	○	×	○	×	○	○	×	×	○	×	○	○	×	×	○	×
構造体	×	×	○	○	×	○	○	×	○	○	○	○	×	×	×	×	○	○	×	×	×	×	○	○
書式付													○	×	○	×	○	○	○	×	○	×	○	○

3. 測定に用いた計算機，コンパイラおよびコンパイラオプション

用いた計算機は富士通製 SPARC Enterprise M9000，FX1 および HX600 である．各計算機の諸元を表 2 に示す．

表 2. 各計算機の諸元

	M9000	FX1	HX600
CPU	SPARC64VII		AMD Opteron (Shanghai)
1CPU あたりのコア数	4		
クロック周波数	2.5GHz		
コアあたりの理論演算性能	10GFlops		
L1 キャッシュ	64KB(データキャッシュ，コア毎)		
L2 キャッシュ	6MB(コア共通)		512KB(コア毎)
L3 キャッシュ	なし		6MB(コア共通)
OS	Solaris10	OpenSolaris	RHEL4.7

また，用いたコンパイラとコンパイラオプションは下記の通りである．これらのオプションを指定することにより，いずれのコンパイラでも最高レベルの最適化が行なわれる．

[C 言語]

富士通コンパイラ

M9000(Version 5.8)	fcc -Kfast,V9 -O5
HX600(Version 3.2)	fcc -Kfast -O5
FX1(Version 5.8)	fcc -Kfast,V9 -O5
SunStudio12	cc -fast
GNU コンパイラ(Version 3.4.6)	gcc -O3

[C++]

富士通コンパイラ

M9000(Version 5.8)	fcc -Kfast,V9 -O5
HX600(Version 3.2)	fcc -Kfast -O5
FX1(Version 5.8)	fcc -Kfast,V9 -O5
SunStudio12	CC -fast
GNU コンパイラ(Version 3.4.6)	g++ -O3

[Fortran]

富士通コンパイラ

M9000(Version 8.1)	fcc -Kfast,V9,tl_trt -X9 -NRtrap -O5
HX600(Version 3.2)	fcc -Kfast -X9 -NRtrap -O5
FX1(Version 8.1)	fcc -Kfast,V9,tl_trt -X9 -NRtrap -O5
SunStudio12	f95 -fast
GNU コンパイラ(Version 3.4.6)	g77 -O3

なお，富士通コンパイラ，あるいは SunStudio12 では，-Kfast あるいは-fast オプションの指定により，コンパイルを行なう計算機のアーキテクチャに最適なオプションが自動的に設定される．富士通コンパイラの-Kfast オプションおよび SunStudio12 の-fast オプションは各計算機で下記のように展開された．

富士通コンパイラの-Kfast オプションの展開結果

[M9000]
-O5 -Kfsimple -Kdalign -Kns -Kfuse -Kmfunc -Kprefetch -VIS2 -FMADD -Keval -KSPARC64VII

[HX600]
-O3 -Komitfp -Keval -Kmfunc -Kprefetch -KSSE2 -KSSE3 –KOPTERON

[FX1]
-O5 -Kfsimple -Kdalign -Kns -Kfuse -Kmfunc -Kprefetch -VIS2 -FMADD -Keval -KSPARC64VII

SunStudio12 の-fast オプションの展開結果

[M9000]
-xO5 -xarch=sparcfmaf -xcache=64/64/2:6144/256/12 -xchip=sparc64vi -xdepend=yes -xmemalign=8s -fsimple=2 -fns=yes -ftrap=%none -xlibmil -xlibmopt -xbuiltin=%all -dryrun

[HX600]
-xO5 -xarch=amdsse4a -xcache=64/64/2:512/64/16 -xchip=amdfam10 -xdepend=yes -fsimple=2 -fns=yes -ftrap=%none -xlibmil -xbuiltin=%all -nofstore -xregs=frameptr -Qoption CC -iropt -Qoption CC -xcallee64 -dryrun -Qoption ube -xcallee=yes

[FX1]
-xO5 -xarch=sparcfmaf -xcache=64/64/2:6144/256/12 -xchip=sparc64vi -xdepend=yes -xm
emalign=8s -fsimple=2 -fns=yes -ftrap=%none -xlibmil -xlibmopt -xbuiltin=%all -dryrun

なお，それぞれのコンパイラオプションの詳細は下記の URL を参照されたい．
富士通：http://www2.itc.nagoya-u.ac.jp/riyou/tuning.pdf
SunStudio12：http://jp.sun.com/products/software/tools/studio12/documentation/ss12/mr/man1/cc.1.html
GNU：http://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc/Optimize-Options.html#Optimize-Options

4. 測定結果

測定結果を図 1～6 に示す．このうち図 1～3 は計算機ごとの CPU 時間を，図 4～6 はコンパイラごとの CPU 時間を示す．また，各図(a)は行列積の計算に要した CPU 時間を，各図(b)は計算結果の書き出しに要した CPU 時間を示す．プログラムの実行は各 10 回行い，CPU 時間の平均値を求めた．

なお，g77 は，Fortrun90 から導入された配列の動的確保に対応していないため，いずれの計算機でもプログラム番号 13,14,17～20,23,24 については測定していない．また，FX1 においては g77 でコンパイルした Fortran プログラムの実行が不可能であったため，測定していない(プログラム番号 13～24)．

図 1～図 6 から観察される事項を下記に示す．

M9000 における計算時間の比較(図 1(a))：

- 富士通コンパイラと SunStudio12 が GNU コンパイラと比較して性能が高い．
- GNU コンパイラで C 言語プログラムにおいて配列を動的に確保した場合には静的に確保した場合(プログラム番号 1,3,5,7)に比べて計算時間が 15%程度増加している．
- C 言語プログラムおよび C++プログラムにおいて配列を 2 次元的に動的確保する(プログラム番号 5,7,11)と，富士通コンパイラおよび SunStudio12 においては，3～4 倍程度，GNU コンパイラにおいては 2 倍程度計算時間が増大する．
- SunStudio12 においては Fortran プログラムで配列を 1 次元配列として確保する(プログラム番号

13～18)と 2 次元的に確保した場合(プログラム番号 19～24)に比べて計算時間が 2 倍程度増大する.

- 富士通コンパイラは配列を 2 次元的に動的確保した場合(プログラム番号 19～24)を除き，安定して計算時間が短い.

M9000 における書き出し時間の比較(図 1(b))：

- いずれのコンパイラでも C 言語プログラムの場合(プログラム番号 1～8)が最も書き出し時間が短く，安定している.
- 書き出し時間の長さは C++プログラム，Fortran プログラム，C 言語プログラムの順番である.
- いずれのコンパイラでも C++プログラム(プログラム番号 9～12)では他の言語に比べて書き出し時間が 5～15 倍増大する.

HX600 における計算時間の比較(図 2(a))：

- 富士通コンパイラと GNU コンパイラにおいては,いずれのプログラムでも同程度の計算時間となった.
- 富士通コンパイラにおいて，配列を構造体のメンバーにして 1 次元的に動的確保した場合(プログラム番号 17),計算時間が 3 秒程度になる.これは他のプログラムでの計算時間の 1/5 程度である.
- 配列を 2 次元的に動的確保した場合(プログラム番号 5,7)および C++プログラムの場合(プログラム番号 9～12)を除き,SunStudio12 が富士通コンパイラおよび GNU コンパイラに比べて 1/2～1/10 程度計算時間が短い.
- SunStudio12 では，C 言語プログラムにおいて，配列を 2 次元的に動的確保した場合(プログラム番号 5,7)は，それ以外の方法で配列を確保した場合に比べて計算時間が 2～3 倍増大する.
- SunStudio12 では，C++プログラムの場合(プログラム番号 9～12)は他言語のプログラムに比べて計算時間が増大する.
- SunStudio12 では，Fortran プログラムにおいて配列を構造体のメンバーとして動的確保する(プログラム番号 17,23)と他の方法で配列を確保した場合に比べて計算時間が 5～10 倍程度増大する.

HX600 における書き出し時間の比較(図 2(b))：

- C 言語プログラムと Fortran プログラムでは C++プログラムに比べて書き出し時間が 1/2～1/3 程度である.
- C 言語プログラムではいずれの場合においても書き出し時間が同程度である.
- Fortran プログラムでは,書式付きの書き出しの方が書式無しの書き出しに比べて書き出し時間が短い.

FX1 における計算時間の比較(図 3(a))：

- C 言語プログラムにおいて配列を 2 次元的に動的確保した場合(プログラム番号 5,7)，C++プログラムで配列を構造体(クラス)のメンバーとして 2 次元的に動的確保した場合(プログラム番号 11)および GNU コンパイラで C++プログラムをコンパイルした場合(プログラム番号 9～12)を除き，いずれにコンパイラおよびプログラムでの比較的計算時間が短い.

FX1 における書き出し時間の比較(図 3(b))：

- C 言語プログラムではいずれの場合においても書き出し時間が同程度である.
- Fortran プログラムでは，書式付きの書き出しの方が書式無しに比べて書き出し時間が短い.

富士通コンパイラにおける計算時間の比較(図 4(a))：

- 全体的な傾向として M9000 と FX1 は HX600 に比べて計算時間が 1/3～1/5 程度短い.
- C 言語プログラムにおいては，ほぼ HX600，M9000，FX1 の順番に計算時間が長いが，配列を 2 次元的に動的確保した場合(プログラム番号 5,7)では M9000 の計算時間が増大する.

富士通コンパイラにおける書き出し時間の比較(図 4(b))：

- いずれの計算機でも C 言語プログラムでの書き出し時間は同程度である.

- HX600 では C 言語プログラムおよび Fortran プログラムでの書き出し時間が同程度である.
- M9000 と FX1 では C 言語プログラムでの書き出し時間は Fortran プログラムに比べて 1/3～1/2 程度である.

SunStudio12 における計算時間の比較(図 5(a)):

- M9000 と FX1 において，C 言語プログラムで配列を 2 次元的に動的確保した場合(プログラム番号5,7)および C++プログラムで配列を構造体(クラス)のメンバーとして 2 次元的に動的確保した場合(プログラム番号 11)は計算時間が増大する.

SunStudio12 における書き出し時間の比較(図 5(b)):

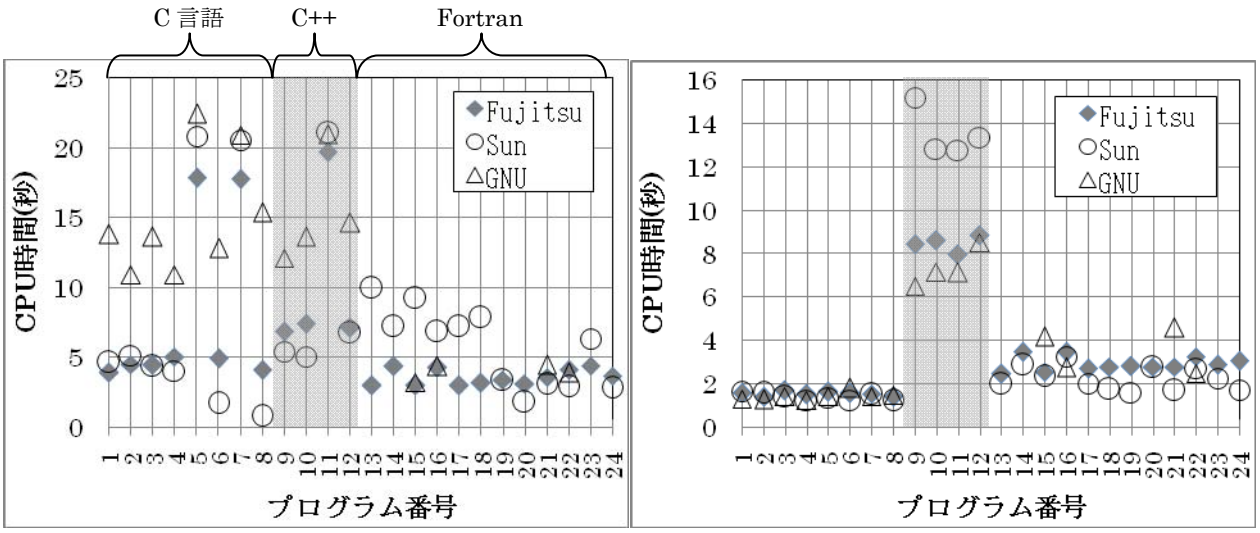
- C 言語プログラムおよび Fortran プログラムは C++プログラムに比べて書き出し時間が短い.
- HX600 では C 言語プログラムおよび Fortran プログラムでの書き出し時間が同程度である.
- いずれの計算機でも, Fortran プログラムで配列を構造体のメンバーとして書式付きで書き出した場合(プログラム番号 14,16,20,22)は，他の Fortran プログラムに比べて書き出し時間が 2 倍程度増大する.

GNU コンパイラにおける計算時間の比較(図 6(a)):

- C 言語プログラムにおいては，ほぼ HX600，M9000，FX1 の順番に計算時間が長いが，配列を 2 次元的に動的確保した場合(プログラム番号 5,7)では M9000 の計算時間が増大する.
- C++プログラム(プログラム番号 9～12)では，いずれの計算機でも同程度の計算時間である.
- C++プログラムで配列を 2 次元的に動的確保した場合(プログラム番号 11)では M9000 の計算時間が増大する.

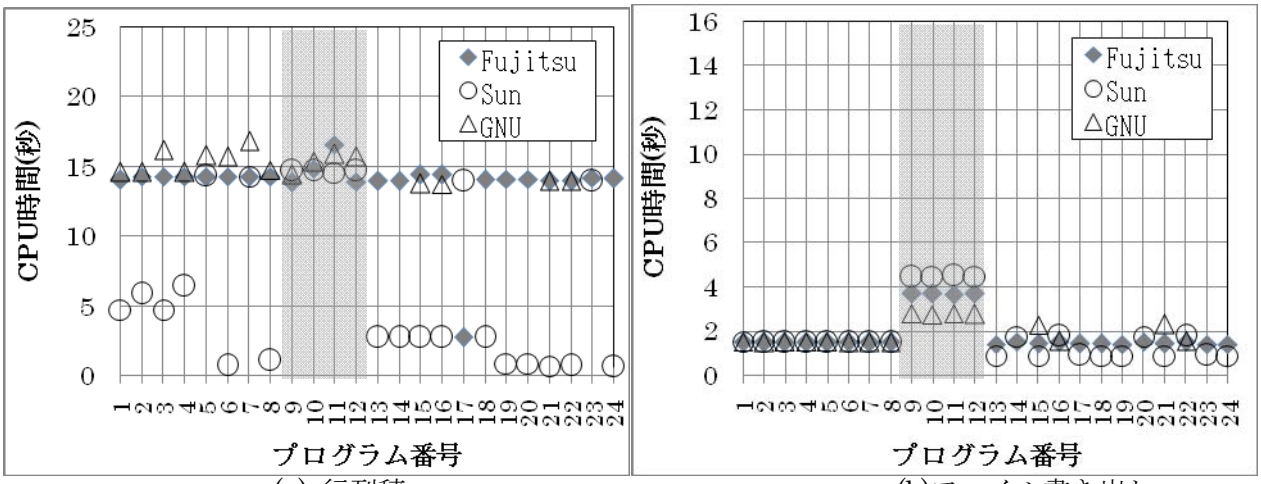
GNU コンパイラにおける書き出し時間の比較(図 6(b)):

- 書き出し時間の長さは C++プログラム，Fortran プログラム，C 言語プログラムの順番である.
- C 言語プログラムでは, いずれの計算機でも書き出し時間はほぼ同程度であるが, C++プログラムと Fortran プログラムでは M9000 の書き出し時間が増大する傾向にある.



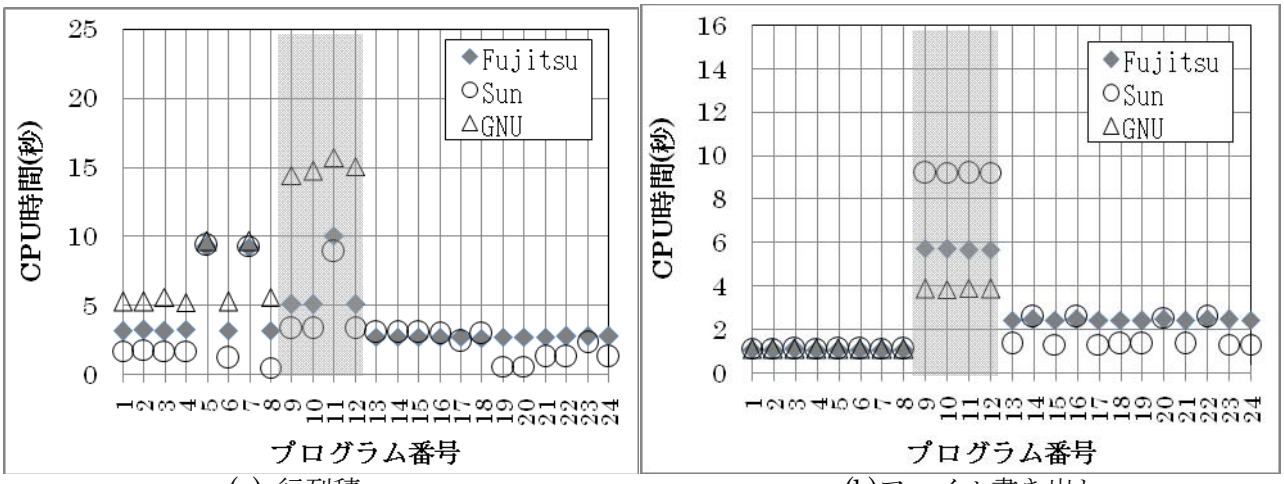
(a) 行列積　　(b)ファイル書き出し

図 1. M9000



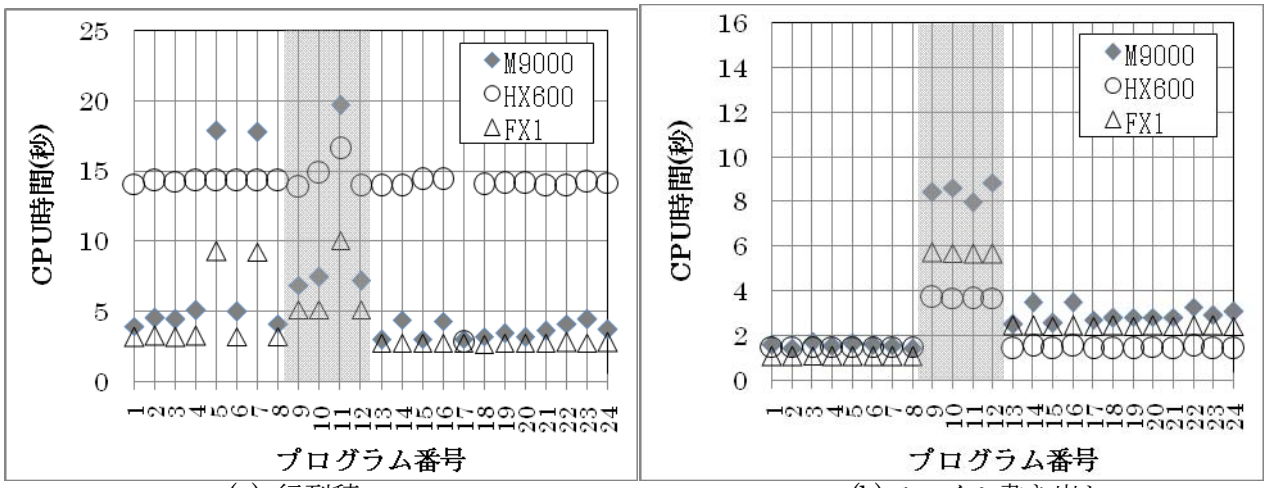
(a) 行列積　　(b)ファイル書き出し

図 2. HX600



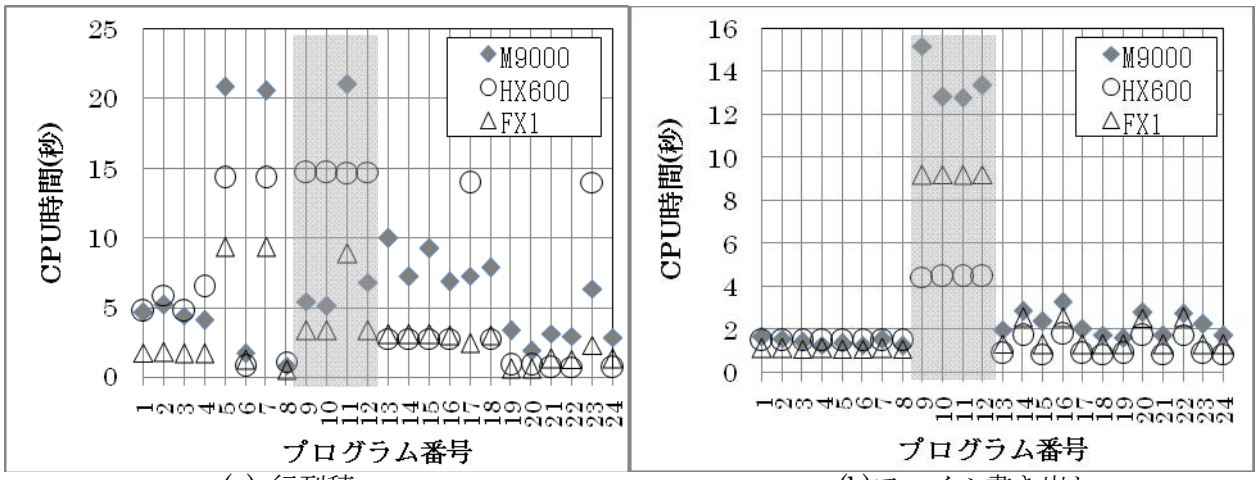
(a) 行列積　　(b)ファイル書き出し

図 3. FX1



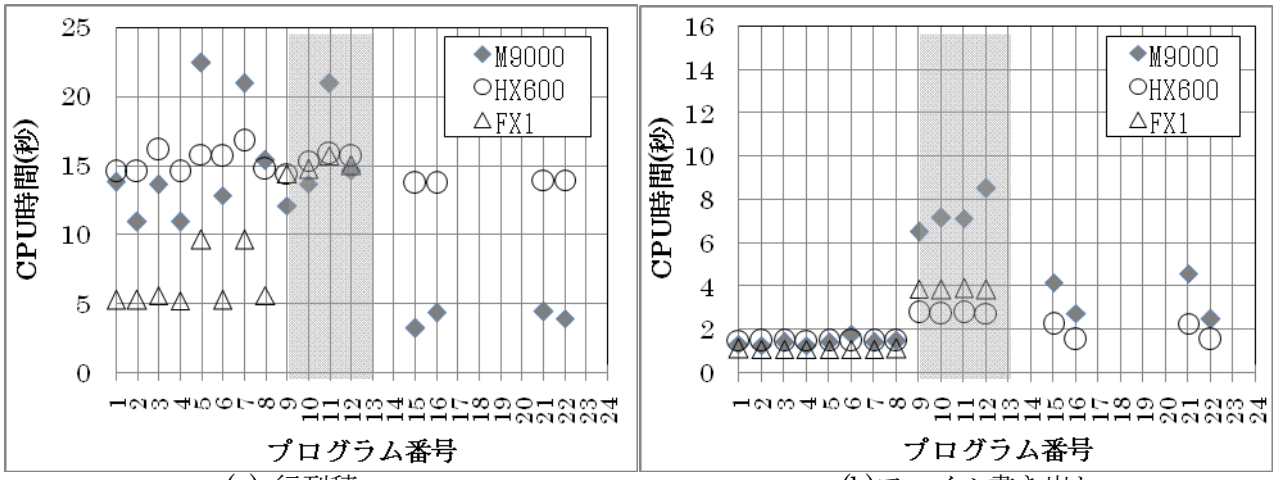
(a) 行列積　　(b)ファイル書き出し

図 4. 富士通コンパイラ



(a) 行列積　　(b)ファイル書き出し

図 5. SunStudio12



(a) 行列積　　(b)ファイル書き出し

図 6. GNU コンパイラ

5. 考察とまとめ

今回の測定において，SunStudio12 が最も高い計算性能を示す場合が多かった．これは，SunStudio12 のインストール時に CPU のアーキテクチャ，キャッシュの特性などを推定し，最適なコンパイルオプションを指定する機構が搭載されているために，いずれの計算機においても高い演算性能を引き出していることが可能であったと推測される．なお，SunStudio12 では，CPU を-xchip=sparc64vi のように誤って推定している．しかし，キャッシュの特性については，-xcache=64/64/2:6144/256/12 のように推定しており，少なくともキャッシュサイズについては SPARC64 VII のキャッシュの特性が正しく推定されている(SPARC64 VI は L1$:128KB および L2$:6144KB, SPARC64 VII は L1$:64KB および L2$:6144KB)．

多くの場合において，FX1 がもっとも高い演算性能を示したことの原因として，名古屋大学情報基盤センターの運用方針では M9000 とは異なり FX1 はデフォルトで逐次プログラムであってもノードを占有して使用できること，および HX600 に比べてメモリバンド幅が広いことが考えられる．また，ノードを占有的に利用することが可能な HX600 と FX1 では，平均値に対する標準偏差の大きさ(=標準偏差/平均値)が高々0.02 程度であるのに対して，ノードを占有できない M9000 では 0.1～2.5 であった．これは M9000 ではメインメモリやローカルディスクへの読み書き時に他のユーザのプログラムの影響によるものと考えられる．

いずれのコンパイラおよび計算機を使った場合でも，C 言語プログラムや C++プログラムで配列を 2 次元的に動的確保した際に計算性能が極端に低下することがある．この原因としては，よく知られている事象ではあるが，実際の値が格納されているメモリ領域に到達するまでにアドレス参照が複数回発生することが原因であると考えられる．一方で C++プログラムを除けば，いずれの言語であっても配列を 1 次元的に確保したり，静的に確保したりすることにより，同程度の計算性能を得られることが分かった．したがって，言語の選択に関しては，2 次元的に動的確保を行なわないという点に注意すれば，C 言語プログラムであっても Fortran プログラムと同程度の演算性能が得られる可能性があることが分かる．

SunStudio12 で Fortran プログラムをコンパイルした場合に，配列を 1 次元的に確保した場合の方が 2 次元的に確保した場合よりも計算時間が長くなる傾向があった．これは富士通コンパイラおよび GNU コンパイラと異なる挙動であるとともに，一般常識とも異なる挙動であり，再検証が必要とされる．

I/O 性能についても，C 言語プログラムと Fortran プログラムに関しては大きな性能の差は見られなかったが，C++プログラムでは大幅に低下した．また，今回の測定では多くの場合において Fortran プログラムでの書式の有無に関して性能の差は見られなかったが，書式付の方が書き出し時間が短い場合も観測された．一般的には書式付の方が I/O 性能が低下すると認識されており，今後の再検証が必要とされる．

以上

【補足資料】
作成したプログラムの例

プログラム番号 7　(C 言語・2 次元的に配列確保・配列の動的確保・構造体の使用)

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define N 1000

typedef struct matrix{
double **mat;
int       x, y;
} Matrix;

int main( void )
{
Matrix     a, b, c;
double tmp;
int       n=N;
int       i, j, k;
time_t tv1, tv2, tv3;
FILE     *output;

output = fopen("mm-2D-struct-dynamic-c.out", "w" );
a.mat = (double **)malloc( sizeof(double *)*N );
b.mat = (double **)malloc( sizeof(double *)*N );
c.mat = (double **)malloc( sizeof(double *)*N );

for( i=0; i<N; i++ ){
a.mat[i] = (double *)malloc( sizeof(double)*N );
b.mat[i] = (double *)malloc( sizeof(double)*N );
c.mat[i] = (double *)malloc( sizeof(double)*N );
  }

for( i=0; i<N; i++ ){
for( j=0; j<N; j++ ){
a.mat[i][j] = (N-i) * j;
b.mat[i][j] = (N-i) * (N-j);
    }
  }

a.x = b.x = c.x = N;
a.y = b.y = c.y = N;
tv1   = clock();

for( i=0; i<a.x; i++ ){
for( j=0; j<a.y; j++ ){
tmp = 0.0;
for( k=0; k<a.y; k++ ){
tmp += a.mat[i][k] * b.mat[k][j];
       }
c.mat[i][j] = tmp;
    }
  }

tv2 = clock();
for( i=0; i<a.x; i++ ){
fprintf( output, "[%d]¥n", i);
for( j=0; j<a.y; j++ ){
fprintf( output, "%d¥t%9.6f¥n", j, c.mat[i][j]);
    }
fprintf( output, "¥n");
  }

tv3 = clock();
fprintf( output, "¥n%9.6f¥t%9.6f¥n", (double)(tv2 - tv1)
/CLOCKS_PER_SEC,          (double)(tv3          -
tv2)/CLOCKS_PER_SEC );

return(0);
}
```

プログラム番号 11　(C++・2 次元的に配列確保・配列の動的確保・クラスの使用)

```
#include<iostream>
#include<fstream>
#include<iomanip>
#include<time.h>
#define N 1000

using namespace std;

class Matrix
{
public:
Matrix(int size);
~Matrix();
void product(Matrix a, Matrix b);
double **matrix;
int x, y;
};

Matrix::Matrix(int size)
{
int i;

matrix = new double*[size];
for( i=0; i<size; i++ ) matrix[i] = new double[size];
x = size;
y = size;
}

void Matrix::product(Matrix a, Matrix b)
{
   double tmp;
   int i, j, k;

   for( i=0; i<x; i++ ){
      for( j=0; j<y; j++ ){
tmp = 0.0;
for( k=0; k<a.x; k++ ){
tmp += a.matrix[i][k] * b.matrix[k][j];
         }
matrix[i][j] = tmp;
      }
   }
}

int main( void )
{
Matrix a(N), b(N), c(N);
int      i, j;
time_t tv1, tv2, tv3;
ofstream output;

output.open("mm-2D-dynamic-c++.out", ios::out);

for( i=0; i<a.x; i++ ){
for( j=0; j<a.y; j++ ){
a.matrix[i][j] = (N-i) * j;
b.matrix[i][j] = (N-i) * (N-j);
      }
   }

tv1 = clock();
c.product( a, b );
tv2 = clock();

output.width(15);
output << setprecision(6);
output << setiosflags(ios::fixed);

for( i=0; i<c.x; i++ ){
      output << "["<< i << "]" << endl;
      for( j=0; j<c.y; j++ ){
         output << j << "¥t" << c.matrix[i][j] << endl;
       }
      output << endl;
   }
   tv3 = clock();

output << endl << (double)(tv2 - tv1)/CLOCKS_PER_SEC << "¥t" << (double)(tv3 - tv2)/CLOCKS_PER_SEC << endl;

   return(0);
}
```

プログラム番号 23　(Fortran・2 次元的に配列確保・配列の動的確保・構造体の使用・書式付き出力)

```
 program main
 type data
    real*8, allocatable ::    matrix(:,:)
    integer    n
 end type data
 type(data) a, b, c
 real*8    tmp
 integer i, j, k
 real       tv1, tv2, tv3

 open(17, file='mm-2D-struct-dynamic-withFormat-f.out',
& status='replace')

 a%n = 1000
 b%n = 1000
 c%n = 1000

 allocate(a%matrix(a%n, a%n))
 allocate(b%matrix(b%n, b%n))
 allocate(c%matrix(c%n, c%n))

 do j=1, a%n
    do i=1, a%n
       a%matrix(i, j) = (a%n-i+1) * (j-1);
       b%matrix(i, j) = (b%n-i+1) * (b%n-j+1);
    enddo
 enddo

call cpu_time( tv1 );

 do i=1, a%n
    do j=1, b%n
       tmp = 0.0;
       do k=1, c%n
         tmp = tmp + a%matrix(i, k) * b%matrix(k, j)
         enddo
      c%matrix(i, j) = tmp
   enddo
enddo

call cpu_time( tv2 );

do j=1, c%n
   write(17, "(i4)") j
   do i=1, c%n
      write(17, "(i4,6x,f20.6)") i,    c%matrix(i, j)
   enddo
enddo

call cpu_time( tv3 )

write(17, *) tv2-tv1, tv3-tv2

stop
end
```

3.2.2. 並列効果の低下とラージページの関係について

富士通株式会社　　内藤 俊也

この報告は、C，C++，Fortran の速度比較において、C 言語の OpenMP の並列効果でみられた性能低下の現象について調査結果を報告するものです。

1. はじめに

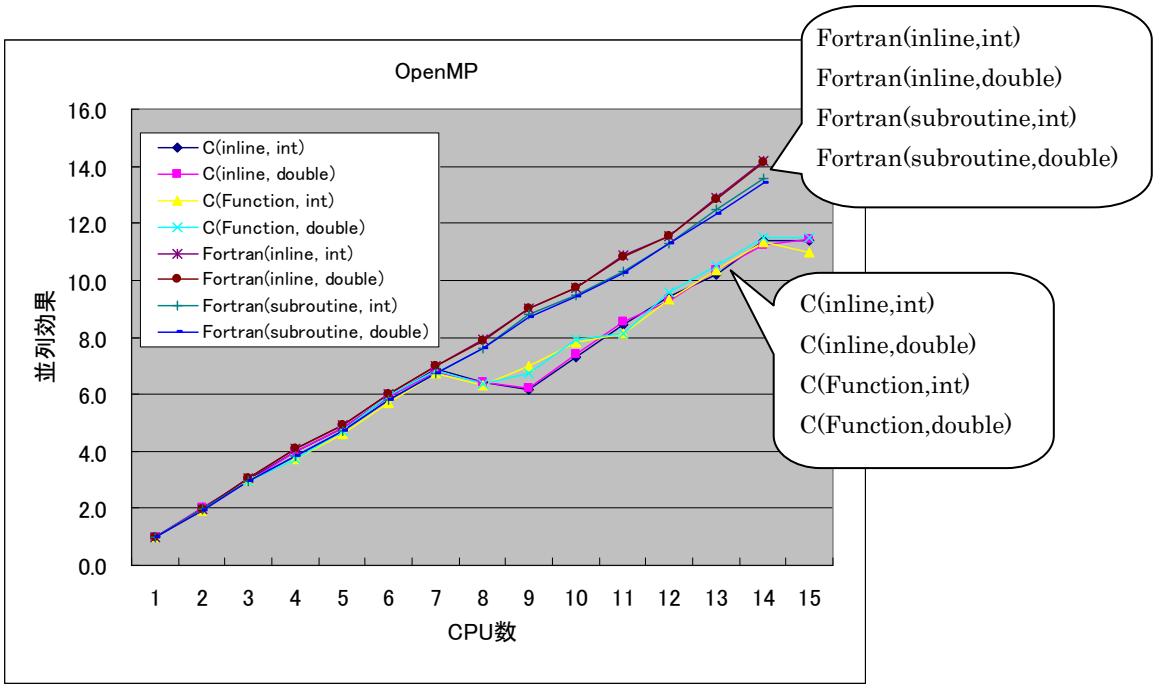
C OpenMP 性能が 8 並列以上で並列効果（**※1**）が低下していることについて、原因を説明します。まず、使用したプログラムの一部を下記に示します。

プログラムの一部(C 言語バージョン)

```
/* Median filter */
for( z=mask_size/2; z<image_size-mask_size/2; z++ ){      /*  このループを並列化  */
  for( y=mask_size/2; y<image_size-mask_size/2; y++ ){
    for( x=mask_size/2; x<image_size-mask_size/2; x++ ){
      i = 0;
      for( zz=-mask_size/2; zz<=mask_size/2; zz++ ){
        for( yy=-mask_size/2; yy<=mask_size/2; yy++ ){
          for( xx=-mask_size/2; xx<=mask_size/2; xx++ ){
            list[i] = image[(x+xx)+(y+yy)*image_size+(z+zz)*image_size*image_size];
            i++;
          }
        }
      }
      /* Bubble sort */
      for( i=0; i<mask_size*mask_size*mask_size-1; i++ ){
        for( j=1; j<mask_size*mask_size*mask_size; j++ ){
          if( list[i]<list[j] ){
            tmp = list[i]; list[i] = list[j]; list[j] = tmp;
          }
        }
      }
        median[x+y*image_size+z*image_size*image_size] = list[mask_size*mask_size*mask_size/2];
    }
  }
}
```

このプログラムは 3 次元画像処理で画像中のスパイクノイズを除去するために比較的頻繁に使われるものです。下記のグラフで、subroutine あるいは Function と表記されたものはメディアンフィルタを関数として実装したもの、inline と表記されたものは関数化しなかったものを表します。また、int あるいは double は配列 image が整数型であるか倍精度浮動小数点型であるかを表します。

以下のグラフと測定値で、4 つの OpenMP　C プログラム（インライン|関数、整数|浮動小数点）は、ほぼ同じ現象であるため、「インライン、整数」を取り上げて説明します。



計算時間（秒）		C				Fortran			
		インライン		関数		インライン		subroutine	
	CPU数	整数	浮動小数点	整数	浮動小数点	整数	浮動小数点	整数	浮動小数点
OpenMP	1	6.490	6.740	7.260	7.470	6.460	6.811	6.218	6.607
	2	3.220	3.370	3.810	3.770	3.245	3.419	3.146	3.410
	3	2.177	2.280	2.433	2.510	2.118	2.227	2.123	2.253
	4	1.620	1.690	1.940	1.990	1.577	1.668	1.630	1.736
	5	1.350	1.402	1.570	1.592	1.315	1.383	1.322	1.406
	6	1.100	1.140	1.272	1.250	1.076	1.134	1.071	1.142
	7	0.941	0.990	1.080	1.090	0.925	0.975	0.921	0.980
	8	1.010	1.050	1.150	1.170	0.818	0.863	0.816	0.870
	9	1.050	1.084	1.040	1.108	0.716	0.754	0.706	0.759
	10	0.890	0.910	0.931	0.941	0.664	0.700	0.655	0.702
	11	0.769	0.789	0.891	0.918	0.595	0.628	0.602	0.646
	12	0.690	0.727	0.780	0.780	0.560	0.590	0.550	0.584
	13	0.635	0.650	0.700	0.711	0.502	0.531	0.498	0.536
	14	0.571	0.600	0.640	0.651	0.456	0.481	0.458	0.493
	15	0.570	0.590	0.660	0.650	–	–	0.452	0.483

※1：並列効果は、clock( )でループの実行時間を計測した CPU 時間を、並列数で割り 1 並列を基準時間（=1.0）として表現しています。ループの時間計測は、omp parallel for 構文の直前と直後で clock( ) を呼び出して区間計測しています。
なお、clock(3C)は、プロセス（全スレッドの合計）の usr+sys 時間を計測します。

2. 性能低下の原因について

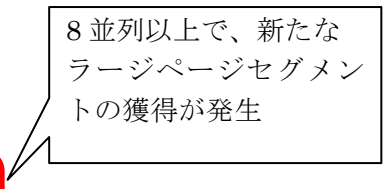
① C OpenMP　8 並列以上での性能低下の原因

1～7 並列と比較して、8 並列以上の場合に新たなラージページセグメントを獲得するコスト（約 0.15 秒～0.2 秒）が発生しており、そのため、性能低下に見えています。これを以下に説明します。

ラージページセグメントの獲得量を NQS (qsub –oi)の統計情報から採取しました。

下の表は、並列数毎に個別のジョブ（計 15 個のジョブ）として実行し、並列数毎のラージページ使用量を計測したものです。「Used Resource」の「Max Large Page」が獲得したラージページセグメントの合計を表しており、8 並列以上の場合に新たなラージページセグメントの獲得が発生しています。

並列数	ラージページ		
	Allocated Resource (MB)	Used Resource Average (MB)	Max(MB)
1	3072	974	1024
2	3072	935	1024
3	3072	947	1024
4	3072	811	1024
5	3072	774	1024
6	3072	807	1024
7	3072	861	1024
8	3072	1425	2048
9	3072	1457	2048
10	3072	1324	2048
11	3072	1343	2048
12	3072	1263	2048
13	3072	1208	2048
14	3072	1205	2048
15	3072	1144	2048



測定したマシンの設定では、スレッド毎のスタック領域（**※2**）が 128MB になっているため、プログラム全体で必要なスタックサイズは、128MB×並列数となります。プログラムが獲得するスタックサイズは、8 並列のときに 1024MB になり、ヒープ域を合わせたラージページ使用量が初期獲得量（1GB）を超えることが計算上でも分かります。このため、8 並列以上の場合に、新たなラージページセグメント獲得のためのコストが発生しています。

新たなラージページセグメント獲得のためのコストは、約 0.15 秒 ～ 0.2 秒であり、実行時間が長ければ無視できるほどのコストであると考えます。今回の 8 並列測定結果は約 1 秒であるため、コストの割合が多く見えていて、グラフではこのコストが顕著に表れています。

今回の報告は OpenMP の調査結果ですが、**自動並列時も同じ現象**になります。

また、測定するマシンの設定により、新たなラージページセグメント獲得の並列数が変わりますので注意して下さい。

※2：スレッド毎のスタック領域は、基本的にプロセスのスタック領域と同じ大きさです。

② C と Fortran の違い(何故 Fortran で発生しないか)

ラージページセグメントはスレッド生成時に獲得されますが、スレッド生成のタイミングが C と Fortran とで異なる場合があります。

C の場合、常に omp parallel ディレクティブが現れた位置にスレッドが生成されます。

Fortran の場合、主プログラム内に OpenMP ディレクティブが存在する場合に特殊な処理を行っています。この処理は主プログラムの先頭にダミーの omp parallel があるかのようなオブジェクトプログラムを生成しますが、このダミーの omp parallel の時点でスレッドが生成され、ラージページセグメントを獲得するコストが Fortran の場合に区間計測の外側で行われることになります。これが C と Fortran の違いになります。

しかし、プログラム全体の計測時間であれば、ラージページセグメントを獲得するコストが C と Fortran 共に含まれるため、**C と Fortran と同様の現象**になります。

3. 補足

本件の報告とは直接関係ありませんが、グラフと測定値より 14 並列と 15 並列で並列効果が変わらないように見えます。これは並列化されるループの回転数が 126 回転であり、14 並列では各スレッド 9 回転、15 並列でも最大 9 回転（8 回転と 9 回転のスレッドが混在）となるためです。

以上

3.2.3. 流体解析から見る Fortran90 の構造体性能評価

宇宙航空研究開発機構　　高木 亮治

1. はじめに

Fortran90 では、構造体、動的配列、ポインターなど様々な便利な機能が追加され、ユーザーがプログラムを作成する際に選択の幅が広がりより便利になった。一方で、実際のアプリケーションプログラムを開発する際には、解析対象となる物理現象を記述する数学モデルやそれらを解析するための計算手法が内包する階層構造を反映したプログラムを作成できるかどうかは一つの重要な観点であると考えられる。Fortran90 で導入された構造体、モジュール、動的配列などの機能はデータ構造の階層化を容易に実現できるものとして非常に便利な機能であるが、従来の静的な配列などに比べて性能面でのデメリットが考えられる。ここでは広く一般の科学技術計算プログラムを作成する際の指針を得ることを最終的な目標として、例えば流体解析プログラムを実装する際に、性能も考慮するとどのような機能を利用してプログラムを実装すれば良いか、逆にこういう実装は駄目ということを確認する。

2. 流体解析プログラム（複合格子法）

流体解析プログラムはデータ構造の観点で整理すると幾つかの種類に分類できるが、ここではマルチブロック構造格子法を対象とする。マルチブロック構造格子は図 1 で示すように、構造格子を 1 つのブロックとして計算空間全体を複数のブロックで構成する手法である（図 2 に例を示す）。特徴として①各ブロックのサイズ、形状はばらばらである、②各ブロック間は非構造的に接合する、③各ブロック内は単一の構造格子である、といった点が挙げられる。マルチブロック構造格子法ではデータ構造に階層構造（①計算空間が複数のブロックで構成される、②各ブロックは構造格子となる、③各格子点で物理量が定義される）が存在する。

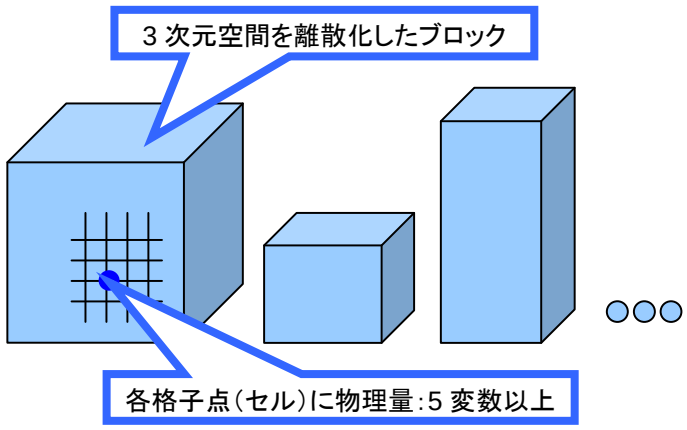


図 1. マルチブロック構造格子のコンセプト

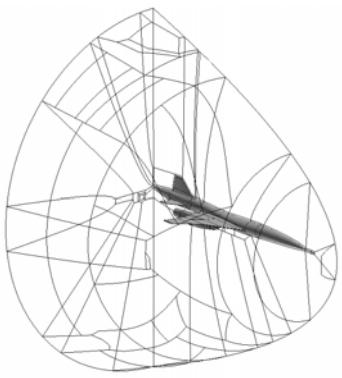


図 1. マルチブロック構造格子の例

特に注意すべき点として各ブロックのサイズ、形状はばらばらであり、これらは解析対象の形状もしくは計算格子作成の都合で決められること、また、汎用的なプログラムとしたいという理由により配列の大きさ、形状は実行時に指定したいという要求があること、である。

3. プログラム概要

3.1 概要

流体解析プログラムでは 3 次元構造格子（I,J,K）で定義されるデータに対して I、J、K 方向それぞれのスイープを実施する。その際にスイープ方向の隣接点を利用した流束(f)の計算を実施し、元の配列(q）の値を更新する。ここでは簡単な場合として 1 次精度の流束（対流項、SHUS スキーム）の計算を考える。高次精度の流束の場合、ステンシル（計算に利用する隣接点の個数）が増加する。ブロック数は 21、各ブロックは 51x51x51 とする。プログラムの概略を以下に示す。

```
I 方向の f および dq の計算
 do j,k,n
 do i=1,imax-1
    f(i,j,k,n) = ...
 enddo
 do j,k,n
 do i=2,imax-1
    dq(i,j,k,n) = dq(i,j,k,n) + f(i,j,k,n) - f(i-1,j,k,n)
 enddo

J 方向の f および dq の計算
 do k,i,n
 do j=1,jmax-1
    f(i,j,k,n) = ...
 enddo
 do k,i,n
 do j=2,jmax-1
    dq(i,j,k,n) = dq(i,j,k,n) + f(i,j,k,n) - f(i,j-1,k,n)
 enddo

K 方向の f および dq の計算
 do i,j,n
 do k=1,kmax-1
    f(i,j,k,n) = ...
 enddo
 do i,j,n
 do k=2,kmax-1
    dq(i,j,k,n) = dq(i,j,k,n) + f(i,j,k,n) - f(i,j,k-1,n)
 enddo

q の更新
do i,j,k,n
   q(i,j,k,n) = q(i,j,k,n) + dq(i,j,k,n)
enddo
```

3.2 データ構造

マルチブロック構造格子を実装するデータ構造として以下の 7 パターンを考えた。

1 構造体を用いて階層構造を表現する場合：ブロック(M)/格子点(I,J,K)/物理量(N)

A：blk(:)%q(:,:,:,:)、インデックスは I,J,K,N（I=J=K=51,N=5）、動的配列%動的配列

B：blk(:)%q(:,:,:,:)、インデックスは N,I,J,K（I=J=K=51,N=5）、動的配列%動的配列

C：blk(:)%q(I,J,K,N)、動的配列%静的配列

D：blk(M)%q(I,J,K,N)、静的配列%静的配列

E：blk(:)%cell(:,:,:)%q(:)、動的配列%動的配列%動的配列

F：blk(:)%cell(:,:,:)%q(N)、動的配列%動的配列%静的配列

2 通常の配列を用いて階層構造を持たない場合：

G：q(I,J,K,N,M)、（I=J=K=51, N=5, M=21）、静的配列

3.3 ループ構造

計算のループとして表 1 で示す 4 種類（カーネル）を考えた。

表 1. カーネル（ループ構造）

カーネル 0	カーネル 1	カーネル 2	カーネル 3
スイープ方向にかかわらず、ループネストの順番を変えない。（インデックスの順番通り） do ndir=1,3 do n=1,5 do k のループ do j のループ do i のループ f の計算 enddo do n=1,5 do k のループ do j のループ do i のループ dq の計算 enddo enddo do n=1,5 do k のループ do j のループ do i のループ q の更新 enddo	カーネル 0 と同じだが、n のループを最内に。 do ndir=1,3 do k のループ do j のループ do i のループ do n＝1,5 f の計算 enddo do k のループ do j のループ do i のループ do n=1,5 dq の計算 enddo enddo do k のループ do j のループ do i のループ do n=1,5 q の更新 enddo	スイープの方向に合わせてループネストの順番を変える。 I 方向 do k=1,kmax do j=1,jmax do i=1,imax-1 f の計算 enddo do n=1,5 do i=2,imax-1 dq の計算 enddo enddo J 方向 do k=1,kmax do i=1,imax do j=1,jmax-1 f の計算 enddo do n=1,5 do j=2,jmax-1 dq の計算 enddo enddo K 方向 do j=1,jmax do i=1,imax do k=1,kmax-1 f の計算 enddo do n=1,5 do k=2,kmax-1 dq の計算 enddo enddo do n=1,5 do k=1,kmax do j=1,jmax do i=1,imax q の更新 enddo	カーネル 2 で n=1,5 のループを最内とする。 I 方向 do k=1,kmax do j=1,jmax do i=1,imax-1 f の計算 enddo do i=2,imax-1 do n=1,5 dq の計算 enddo enddo J 方向 do k=1,kmax do i=1,imax do j=1,jmax-1 f の計算 enddo do j=2,jmax-1 do n=1,5 dq の計算 enddo enddo K 方向 do j=1,jmax do i=1,imax do k=1,kmax-1 f の計算 enddo do k=2,kmax-1 do n=1,5 dq の計算 enddo enddo do k=1,kmax do j=1,jmax do i=1,imax do n=1,5 q の更新 enddo

4. 計測結果

4.1 計測環境

① PC（Intel Core2 Extreme (QX9650)、3.0GHz、X38、DDR3-1333）
コンパイルオプション：-O5

② PC（pointer 属性を allocatable 属性に変更）
コンパイルオプション：-O5

③ FX1（SPARC 64VII, 2.5GHz, visIMPACT なし）
コンパイルオプション：-O5

④ FX1（SPARC 64VII, 2.5GHz, visIMPACT）
コンパイルオプション：-O5 JAXA デフォルト

⑤ FX1
コンパイルオプション：-O5 JAXA デフォルト –x200 –Karray_private
–Kprefetch_strong,noalias=s –Ncalleralloc=2 –Kauto –Kthreadsafe –KNOFILTLD

⑥ FX1（pointer 属性を allocatable 属性に変更）
コンパイルオプション：-O5 JAXA デフォルト –x200 –Karray_private
–Kprefetch_strong,noalias=s –Ncalleralloc=2 –Kauto –Kthreadsafe –KNOFILTLD

JAXA デフォルト：-Kfast,ocl –Nrtrap –Ktl_trt –Kimpact –Kmfunc=2
–Kpreex,prefetch_model=FX1

4.2 計測結果

計測結果を表 2 に示す。PC に関しては連続して 10 回計測を行いその平均値を計測結果とした。FX1 に関しては 1 回の計測値である。念のため手動で数回計測を行ったが、計測値はその都度若干変わったが、傾向としては大きくは変動しなかった。

4.3 考察

・データ構造
  ・A と G および B と G の比較から構造体を使っても大きなデメリットはない。
  ・A と C、D の比較から動的配列よりも静的配列のほうが有利。
  ・PC では pointer（①PC）と allocatable（②PC）の違いは小さい。（allocatable が若干有利？）
  ・FX1 では allocatable（⑥FX1）よりも pointer（⑤FX1）が良い。（本計測を行った時点では富士通コンパイラの allocatable 属性に対する最適化が不十分であったためである）
  ・C,D,G が他に比べて速い（そうでもないケースもあるが）ことから静的配列が有利。
  ・E が最悪、F も全般的に駄目。でも PC では F の 3 が最速となった。原因は不明。
・カーネル
  ・2,3 は駄目。
  ・0 と 1 は大きな差がない（コンパイルリストを確認するとカーネル 0 では do n=1,5（最外側）のループで自動並列化されていた。ディレクティブを挿入することで do n=1,5 を除いて内側のループで自動並列化を行ったが、性能的には大差はなかった。）
・全般
  ・PC と FX1 の比較では PC は比較的ケース間の性能差が小さいが FX1 は大きい。
  ・FX1 のオプションの効果が大。

表 2. 計測結果

データ構造	カーネル	時間［sec］					
		①PC	②PC	③FX1	④FX1	⑤FX1	⑥FX1
A	0	1.052	0.924	1.171	1.114	0.377	4.543
	1	0.947	0.892	1.220	1.271	0.361	4.315
	2	1.380	1.205	2.211	2.382	2.098	7.786
	3	1.289	1.186	2.211	2.882	2.101	7.466
B	0	1.676	1.546	1.447	1.404	0.497	4.589
	1	0.915	0.860	1.241	1.297	0.346	4.173
	2	1.151	1.092	1.767	1.809	1.746	5.547
	3	0.789	0.869	1.774	1.751	1.746	5.243
C	0	0.810	0.830	0.856	0.789	0.296	0.879
	1	0.848	0.828	0.831	0.765	0.271	0.957
	2	0.949	1.051	2.108	2.246	2.015	2.094
	3	0.982	1.025	1.982	2.251	1.978	2.021
D	0	0.858	0.734	0.971	0.788	0.877	0.877
	1	0.820	0.857	0.974	0.771	0.936	0.935
	2	1.031	0.973	2.262	2.216	2.080	2.084
	3	0.989	0.983	2.341	2.244	2.085	2.086
E	0	4.703	4.594	13.268	13.249	6.808	15.692
	1	1.760	1.606	5.154	5.079	3.401	7.575
	2	2.829	2.556	4.052	6.092	3.593	8.590
	3	1.564	1.546	4.030	4.101	3.592	6.362
F	0	1.899	1.913	1.690	1.213	1.216	1.210
	1	0.890	0.881	1.089	1.011	1.013	0.987
	2	0.733	0.645	1.017	1.005	0.992	1.358
	3	0.662	0.626	1.009	1.005	0.982	0.923
G	0	0.795	0.783	0.984	0.792	0.312	0.296
	1	0.812	0.792	0.950	0.766	0.304	0.273
	2	1.053	1.042	2.347	2.229	2.089	2.081
	3	1.036	1.042	2.333	2.229	2.252	2.254

	最短時間より倍以上遅いケース

5. まとめ

・静的配列がやはり性能的に有利だが、構造体もループの回し方に気をつければそれほど大きなペナルティーはない。

・ループは常に同じ順番（インデックスの順番）でまわすべき。

以上

3.2.4. Fortran90 による格子 QCD コードとその性能

広島大学　　中村　純

1. はじめに

Fortran90 は FORTRAN77 の後継であり、新しく導入された module 機能によりユーザーによるデータの自由な型とそれに対する演算の定義が可能となった。このことにより、可読性の向上、バグの可能性の低減、コーディングの容易さがもたらされた。大量の計算機リソースを必要とする格子 QCD 計算の分野では、新しい解析手法やアルゴリズムが次々と提案されるため、この読みやすくミスを起こしにくい Fortran90 のオブジェクト指向型コーディングは非常に重要である。しかし、得られたコードの性能が低くては科学技術計算用には問題である。

ここでは、Fortran90 によりどのようにコードが書かれるのか、またそのコードの高速化ではどのような点に配慮するべきかについて検討する。

```
      TYPE(g_field1)  staple, temp1, temp2, temp3

c           x+nu temp2
c            .---------.
c            I         I
c   temp1 I             I
c            I         I
c            x        x+mu

             temp1 = u(nu)
             temp2 = nu.gshift.u(mu)
             temp3 = temp1 * temp2
             temp1 = mu.gshift.u(nu)
             staple = staple + (fac*(temp3.prodAD.temp1))
```

ここで、TYPE(g_field1) はあらかじめ以下のように定義されている。

```
      TYPE g_field1
         SEQUENCE
         COMPLEX*16, DIMENSION(NC,NC,NV/NBUSH) :: g
         INTEGER parity, direction
      END TYPE
```

また上記のプログラム中の演算（=, +, .gshift. , .prodAD）も同様にモジュール中で定義されている。例えば加法は、以下のように演算子「+」に関数 gadd を対応させて定義されている。

```
      ....
      INTERFACE OPERATOR(+)
         MODULE PROCEDURE gadd
      END INTERFACE
      ....
c-----------------------------------------------------------------------c
      FUNCTION gadd(a,b) RESULT(c)
c-----------------------------------------------------------------------c
      TYPE(g_field1), INTENT(IN):: a, b
      TYPE(g_field1) c

      do i = 1, NV/NBUSH

         c%g( 1, 1, i) = a%g( 1, 1, i) + b%g( 1, 1, i)
         c%g( 1, 2, i) = a%g( 1, 2, i) + b%g( 1, 2, i)
         c%g( 1, 3, i) = a%g( 1, 3, i) + b%g( 1, 3, i)
      .........

      enddo

      END FUNCTION
```

これらのモジュールで定義された演算も、コンパイラーが計算式の中の変数の型で演算を区別することによって、通常の演算同様適切に行われる．一度演算を定義すると、上記のように通常の演算と同じようにプログラム中で使用することができるので，紙の上での式とコード中の式の距離が短い。

上のコードは、かつては以下のような形で書かれていた。

```
call   movsites(jd,nvol,site(1,1),site(1,3))

call   getlinks(site(1,1),temp2,jd,nvol)
call   getlinks(site(1,3),temp3,id,nvol)
call   prodlink(temp2,temp3,temp4,nvol,1)

call   getlinks(site(1,2),temp2,jd,nvol)
call   prodlink(temp4,temp2,temp3,nvol,3)
call   prodlink(temp1,temp3,temp2,nvol,3)

call   addlink(plaq,temp2,plaq,nvol)
```

これと比較すると module のありがたみが分かる。

2. Fortran90 と計算スピード

Fortran を使う以上、計算スピードが高いことを期待するのは当然であろう。しかし、これまで国産コンパイラーで疑問に感じることが何度かあった。

- SR8800 において、module で定義した変数が引数に現れるとサブルーティンコールに異常に時間のかかることがあった。
- SX-5 において、module の演算を定義する関数の中の DO-ループは自動並列化の対象にならない。
- Fujitsu コンパイラーでは module で定義した演算は遅いのでは。

3. QCD と HPC

QCD(Quantum Chromodynamics, 量子色力学)の数値シミュレーションは大きな計算機リソースを必要とする。現在、その大部分は大規模疎行列の逆を求めるための計算で CG(Conjugate Gradient, 共役勾配法)系のアルゴリズムが使われている。CG 法は線形方程式

A X = b

の解法であるが、行列とベクトルの積 Y=A X とベクトルの内積 <X|Y>、それにベクトルの和とスカラー倍で表され、A が大規模疎行列の時に強力な手法となる。計算リソースという観点からは行列とベクトルの積が高速化されればよい。

3.1. QCD に現れる行列の形

行列 D を格子 QCD の業界用語で書くと

$$D = I - \kappa \sum_{\mu=1}^{4} (r - \gamma_\mu) U_\mu(x) \delta_{x',x+\hat{\mu}} + (r + \gamma_\mu) U_\mu^\dagger(x') \delta_{x',x-\hat{\mu}}$$

κ ： ホッピング・パラメータ
r ： Wilson 項
γ_μ ： Dirac のガンマ行列(4x4)
$\hat{\mu}$ ： 格子間隔の大きさを持った mu 方向の 4 次元ベクトル
$U_\mu(x)$ ： SU(3) 行列
$U_\mu(x)\dagger$ ： エルミート共役行列（複素・転置）

なので $\vec{Y} = D\vec{X}$ は

$$Y(x) = X(x) - \kappa \sum_{\mu=1}^{4} \left\{ (r - \gamma_\mu) U_\mu(x) X(x + \hat{\mu}) + (r + \gamma_\mu) U_\mu^\dagger(x - \hat{\mu}) X(x - \hat{\mu}) \right\}$$

となる。もう少し普通の人に分かるような表現は、フェルミオン行列は、カラー、Dirac、座標の添え字を顕わに書いて、以下となる。

$$D_{\alpha\beta}^{ab}(x, x') = \delta_{ab}\delta_{\alpha\beta}\delta_{x,x'} - \kappa \sum_{\mu=1}^{4} \left\{ (r - \gamma_\mu)_{\alpha\beta} U_\mu(x)^{ab} \delta_{x',x+\hat{\mu}} + (r + \gamma_\mu)_{\alpha\beta} U_\mu^\dagger(x')^{ab} \delta_{x',x-\hat{\mu}} \right\}$$

ここで $x \pm \hat{\mu}$ は、μ(＝1,2,3,4) 方向の隣の格子点。

$\vec{Y} = D\vec{X}$ は成分を顕わに書けば、以下となる。

$$Y_\alpha^a(x) = X_\alpha^a(x) - \kappa \sum_{\mu=1}^{4} \sum_{\beta=1}^{4} \sum_{b=1}^{3} \left\{ (r - \gamma_\mu)_{\alpha\beta} U_\mu(x)^{ab} X_\beta^b(x + \hat{\mu}) + (r + \gamma_\mu)_{\alpha\beta} U_\mu^\dagger(x - \hat{\mu})^{ab} X_\beta^b(x - \hat{\mu}) \right\}$$

4. 理研でのチューニング

理研の RSCC を使わせていただくときに、チューニングを依頼した。青山幸也氏が中村の計算機コードを解析し(2005 年 10 月)、主なアドバイスは以下の 2 点であった。この結果、小規模な問題で 6.56 秒が 1.03 秒に短縮された。

- キャッシュミスの可能性を低めるために

```
do ic = 1, Nc
  do it = 1, Nt
  do iz = 1, Nz
  do iy = 1, Ny
  do ix = 1, Nx
    b%f(ic,ix,iy,iz,it,1) = ....
  enddo
  enddo
  enddo
  enddo
enddo
```

のループの順番を変更し

```
do it = 1, Nt
do iz = 1, Nz
do iy = 1, Ny
do ix = 1, Nx
  do ic = 1, Nc
    b%f(ic,ix,iy,iz,it,1) = ....
  enddo
enddo
enddo
enddo
enddo
```

とすること（但しコンパイラーが自動的に行う場合もある)、

- module で定義される演算を subroutine で書くこと

その後、コンパイラーのバージョンアップなどもあったはずなので、以下のように一部を module の演算から subroutine にして時間を測定したところ､行列とベクトルの積が 1.65 秒から 1.05 秒になったので、やはり module の演算は subroutine より遅い。

```
      ! ...   the iteration starts

      CALL clock(cpu1,0,2)

      do i = 1, imax

        ! ...   q = W * p
        if(iflag==1)   then
*           q = wxvect(p,2)
            CALL xwxvect(p,q,2)
        else if(iflag==2)   then
*           q = wxvect(p,3)
            CALL xwxvect(p,q,3)
        endif

        ......

      enddo

      CALL clock(cpu2,0,2)
      WRITE(*,*) "cpu: ", cpu2-cpu1

      ....

      END
c-----------------------------------------------------------------------c
      SUBROUTINE xwxvect(x,wxvect,iflag)
c-----------------------------------------------------------------------c
c     wxvect = x              for iflag=1
c              W*x                       2
c              (W_adj)*x                 3
c-----------------------------------------------------------------------c

          ......

          do nu = 1,4

*         temp1 =     nu   .fshift. x
          CALL xfshift(nu,x,temp1)

*         temp2 = (-nu) .fshift. x
          CALL xfshift(-nu,x,temp2)

*         temp = temp + ((hopp(nu)*temp1) + (hopm(nu)*temp2))
          CALL xdmul(hopp(nu),temp1,temp1)
          CALL xdmul(hopp(nu),temp2,temp2)
          CALL xfadd(temp,temp1,temp)
          CALL xfadd(temp,temp2,temp)

          .....
```

5. 考察

関数の場合は、戻り値が一旦一時的な領域にコピーされるため、本コードのようにその本体が巨大な配列である場合はある程度はロスがあるのはやむを得ないと思われる（【補足資料】モジュール内関数とサブルーチンの性能差について 参照）。また、最適化が個々の演算の中で閉じてしまうので、

```
a = b + (s*c)
```

で　a, b, c が大きな配列、s がスカラーの場合など右辺全体に渡る最適化が難しくなる。

今後、Fortran90 の大きな利点である module による演算定義を、数値計算の高速化の阻害を最小限にするように行うことが重要である。このためには、いろいろなコーディング例が公開され、検討されることが望ましい。

以上

【補足資料】モジュール内関数とサブルーチンの性能差について

富士通株式会社　　鈴木 清文

「Fortran90 による格子 QCD コードとその性能」で報告されているモジュール内関数とサブルーチンの性能差について詳細を報告する。

1. 現象

モジュール内で定義された演算（利用者定義関数）をサブルーチン形式に書き換えると高速化される。以下に例を挙げる。

【構造型の定義】

```
      TYPE f_field
        SEQUENCE
        COMPLEX*16, DIMENSION(NC,0:NX+1,0:NY+1,0:NZ+1,0:NT+1,4) :: f
                          ! (Color, x,      y,      z,      t, Dirac)
      END TYPE
```

ここで、NC=3, NX=4, NY=4, NZ=4, NT=4。

【関数の型の定義】

```
      FUNCTION fadd(a,b) RESULT(c)
c-------------------------------------------------------------------------c
      TYPE(f_field), INTENT(IN):: a, b
      TYPE(f_field) c
```

【サブルーチンのインターフェースの定義】

```
      SUBROUTINE Sub_fadd(a,b,c)
c-------------------------------------------------------------------------c
      TYPE(f_field), INTENT(IN):: a, b
      TYPE(f_field), INTENT(OUT):: c
```

2. 関数の方が遅い原因

関数で書いた方が遅い原因は、「Fortran90 による格子 QCD コードとその性能」で指摘されている通り、関数内で関数結果を保持する領域（上記の例では RESULT の c）から関数を呼び出した手続き内の変数にその関数結果をコピーする処理がオーバーヘッドとして見えるためである。上記の f_field という構造型の場合、15552 要素=248832 バイトのコピーが関数から復帰する時に暗黙の内に実行される。

【呼出元手続きにおける処理】

```
  ～=fadd(～) ～～      →     関数結果域=fadd(～)   ★ここでコピー実行
                              ～=関数結果域 ～～

                        コンパイラは関数結果域を生成し、右のように命令を展開する
```

関数は式の途中に記述することができるため、関数結果を一旦受け取るための領域が必要となる。

サブルーチンの形で書かれた場合は、引数として渡された呼び出し元手続きの変数に直接計算結果を格納する形になるため、上記のオーバーヘッドがないことになる。

関数結果のコピーをコンパイラが暗黙の内に実行するという動作は関数の実行論理上必要なものである。関数結果として配列や配列を含む構造型の場合は、その大きさによりコピーのオーバーヘッドが目に見える形で現れてくるため、上記の特性を踏まえて使い分けるような対処が必要となる。

以上

FUJITSU

3.3. MPI

3.3.1. MPIの基本的通信と入出力

富士通株式会社　　志田 直之

Contents

MPI概要

- MPIの歴史
- MPIライブラリの系譜

MPIの歴史

1992/04 MPI Forum設立

1994/10 MPI forumよりMPI-1.0がリリースされる。
MPIライブラリの基本的な128関数をサポート。

1995/07 MPI forumよりMPI-1.1がリリースされる。

1995/07 MPI forumよりMPI-2.0(MPI-1.2を含む)がリリースされる。
MPI-IO・動的プロセス生成・片側通信・ Fortran/C++対応などが追加。

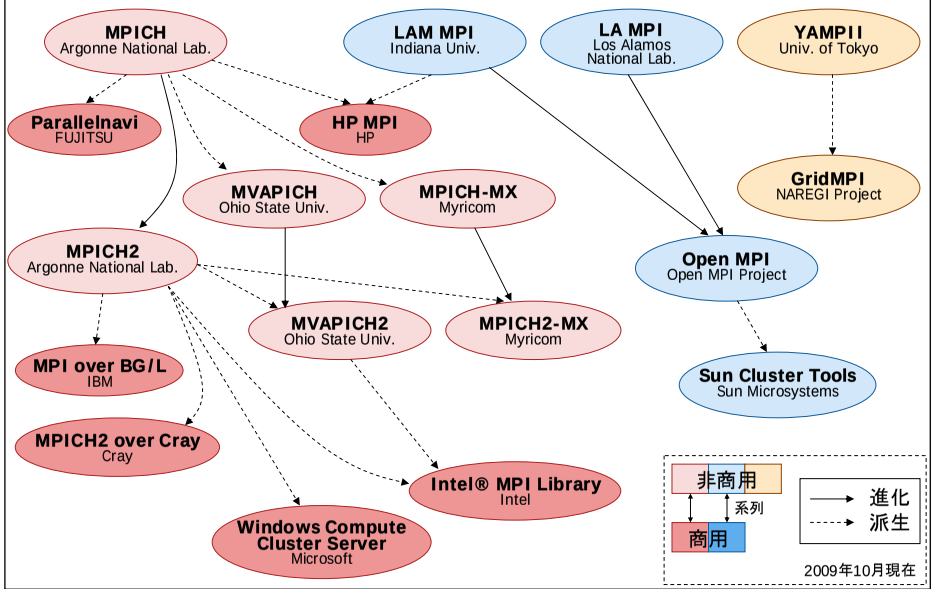
MPI-1.2の分類

1対1通信	39
集団通信	16
派生データ型	13
コミュニケータ	30
プロセス・トポロジー	16
環境管理	14

2008/09 MPI forumよりMPI-2.1がリリースされる。
MPI-2.0の曖昧だった規格を修正する

現在、MPI-2.2, MPI-3.0が検討されている。
MPI-2.2: MPI-2.0に若干の仕様追加を行う。
MPI-3.0: 将来のMPIに向けて、大きな仕様変更を予定。

MPIライブラリの系譜



1対1通信

- ブロッキング通信とノンブロッキング通信
- デッドロック
- 4つの通信モード
- 1対1通信プロトコル

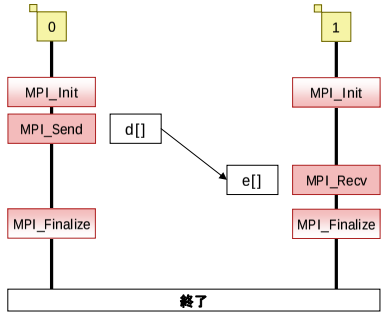
1対1通信とは

- 1対1通信とは、送信と受信が一対となり、通信を行います。

test.c

```
#include "mpi.h"
main(int argc, char *argv[])
{
    int size, rank;
    double d[10], e[10];
    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    switch(rank){
    case(0):
        MPI_Send(d, 10, MPI_DOUBLE, 1,
1000, MPI_COMM_WORLD); //to 1
        break;
    case(1):
        MPI_Recv(e, 10, MPI_DOUBLE, 0,
1000, MPI_COMM_WORLD, &status); // from 0
        break;

    }

    MPI_Finalize();
}
```

```
$ mpicc test.c
$ mpiexec -n 2 -hostfile mpd.hosts ./a.out
```



1対1通信の種類

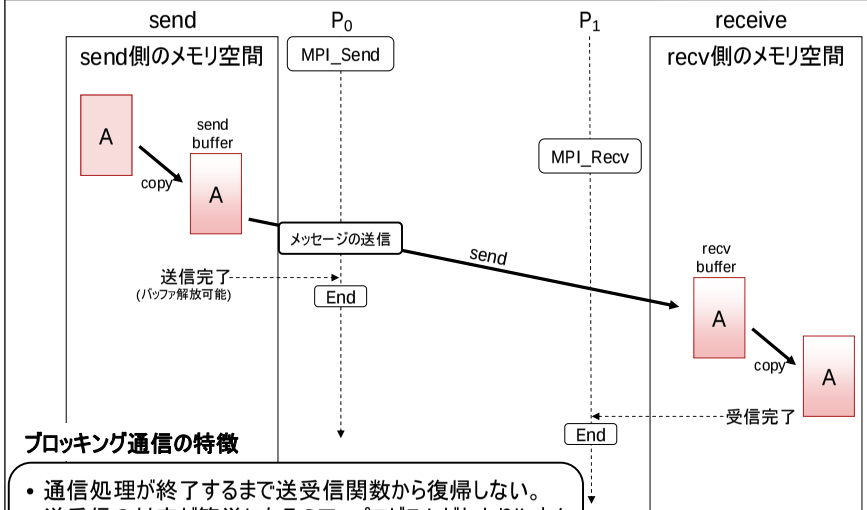
- 1対1通信のAPI

	ブロッキング		ノンブロッキング	
	送信	受信	送信	受信
標準モード	MPI_Send	MPI_Recv	MPI_Isend	MPI_Irecv
バッファモード	MPI_Bsend		MPI_Ibsend	
同期モード	MPI_Ssend		MPI_Issend	
レディモード	MPI_Rsend		MPI_Irsend	
送受信	MPI_Sendrecv			
	MPI_Sendrecv_replace			

- ブロッキング
  - 送信側=送信バッファを解放しても良いタイミング
  - 受信側=受信バッファを解放しても良いタイミング

  になるまで、送信関数・受信関数から復帰しません。すなわち、送信の場合はインタコネクト上にデータがすべて乗った状態、受信の場合はインタコネクトからデータをすべて受け取ったデータが実メモリに書き込まれたことが保証されます。
- ノンブロッキング
  - 送信処理・受信処理を開始する宣言のみで、送信関数・受信関数から復帰します。実際のデータの同期は、MPI_Test関数やMPI_Wait関数などを利用して、ユーザー自身が保証する必要があります。

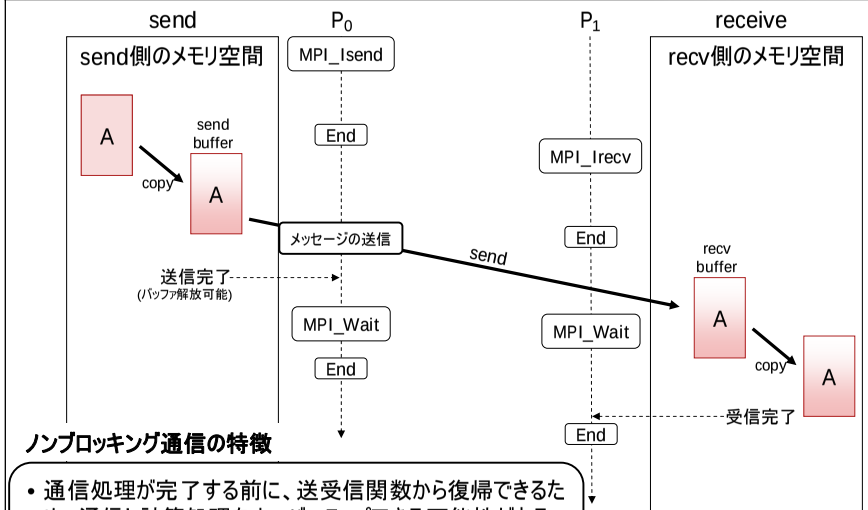
ブロッキング通信



ブロッキング通信の特徴

- 通信処理が終了するまで送受信関数から復帰しない。
- 送受信の対応が簡単になるので、プログラムがわかりやすくなる。
- プログラムの設計次第で、デッドロックの危険性が常にある。

ノンブロッキング通信



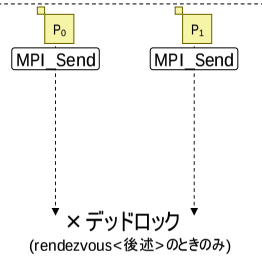
ノンブロッキング通信の特徴

- 通信処理が完了する前に、送受信関数から復帰できるため、通信と計算処理をオーバーラップできる可能性がある。
- ユーザー責任で、ノンブロッキング通信の完了を確認する必要がある。

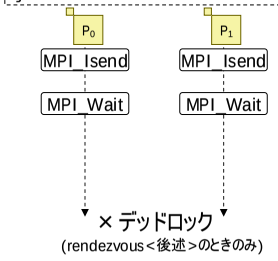
デッドロック

- 送受信処理が待ち状態のため、完了できない状態
  - デッドロックするプログラムは?

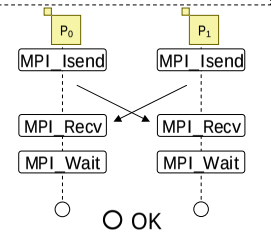
```
switch(RANK){
case 0:
  MPI_Send(To=1,TAG=100);
  MPI_Recv(To=1,TAG=101);
  break;
case 1:
  MPI_Send(To=0,TAG=101);
  MPI_Recv(To=0,TAG=100);
  break;
}
```




```
switch(RANK){
case 0:
  MPI_Isend(To=1,TAG=100);
  MPI_Wait();
  MPI_Recv(To=1,TAG=101);
  break;
case 1:
  MPI_Isend(To=0,TAG=101);
  MPI_Wait();
  MPI_Recv(To=0,TAG=100);
  break;
}
```



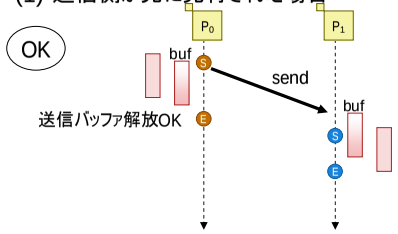

```
switch(RANK){
case 0:
  MPI_Isend(To=1,TAG=100);
  MPI_Recv(To=1,TAG=101);
  MPI_Wait();
  break;
case 1:
  MPI_Isend(To=0,TAG=101);
  MPI_Recv(To=0,TAG=100);
  MPI_Wait();
  break;
}
```



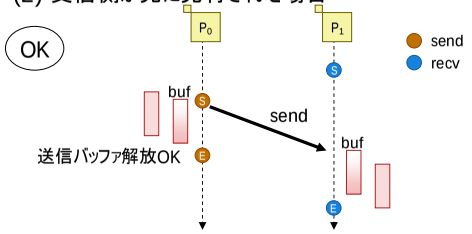
4つの送信モード(1/4)

- 標準モード
  - 対応する受信の順番に関わらず、送信処理を開始することができる。
  - 送信するメッセージがバッファリングされるかどうかは、処理系依存となる。
  - 送信バッファが再利用可能になった場合に、送信処理を完了することができる。
    - メッセージが受信側でバッファリングされる場合
      対応する受信関数が発行される前に、送信関数を復帰することができる。
    - メッセージが受信側でバッファリングされない場合
      対応する受信関数が発行されると、送信関数を復帰することができる。

(1) 送信側が先に発行された場合



(2) 受信側が先に発行された場合

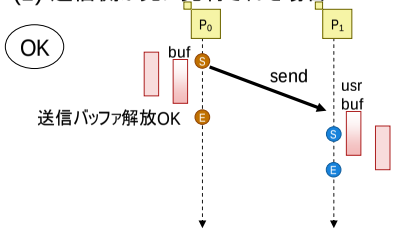


4つの送信モード(2/4)

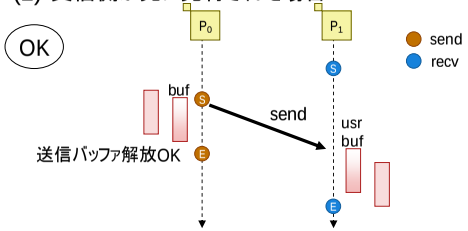
FUJITSU

- バッファモード
  - 対応する受信の順番に関わらず、送信処理を開始することができる。
  - 送信するメッセージは、受信側にバッファリングされる。受信側のバッファアロケーションは、ユーザーが行う必要がある。(MPI_Buffer_attach)
  - 受信バッファに十分な大きさがない場合、送信関数はエラーとなる。
  - 送信バッファが再利用可能になった場合に、送信処理を完了することができる。
  - 送信処理は、対応する受信が発行されなくても、送信処理を完了することができる。

(1) 送信側が先に発行された場合



(2) 受信側が先に発行された場合

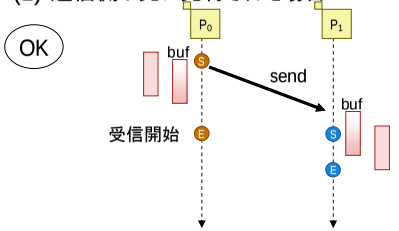


4つの送信モード(3/4)

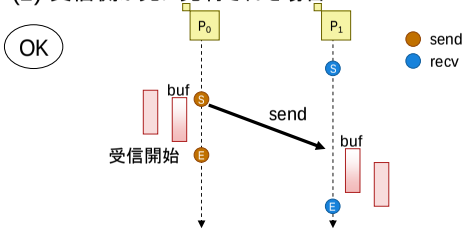
FUJITSU

- 同期モード
  - 対応する受信の順番に関わらず、送信処理を開始することができる。
  - 送信処理は、対応する受信が発行され、さらに、受信操作が開始された場合に、完了することができる。

(1) 送信側が先に発行された場合



(2) 受信側が先に発行された場合

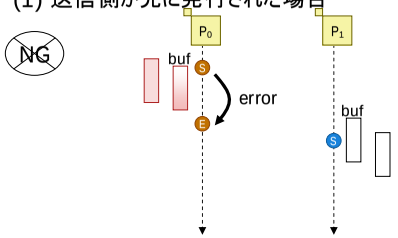


4つの送信モード(4/4)

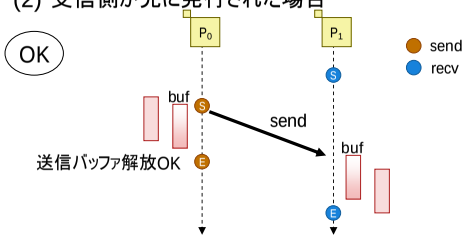
FUJITSU

- レディ通信モード
  - 対応する受信が先に発行されている場合のみ、送信処理を開始することができる。それ以外の場合は、エラーとなる。

(1) 送信側が先に発行された場合

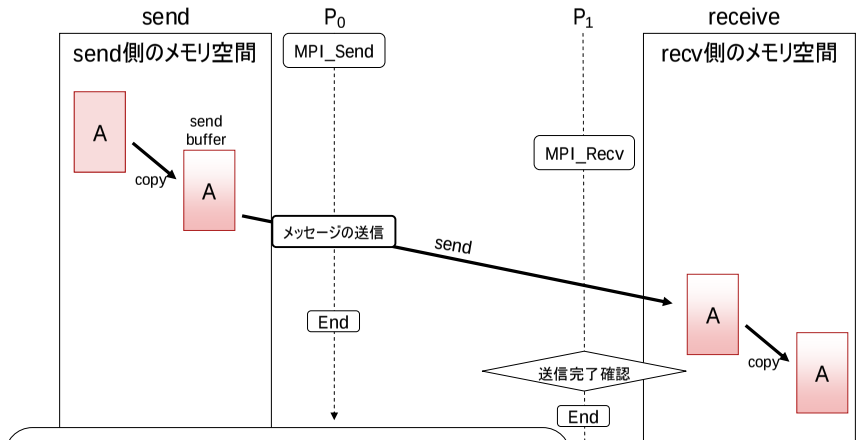


(2) 受信側が先に発行された場合



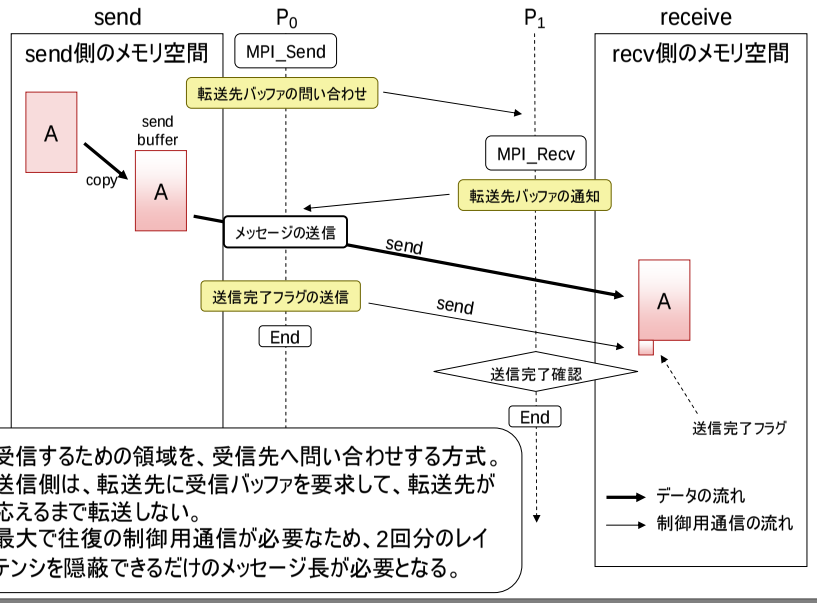
1対1通信プロトコル(eager send/recv)

FUJITSU



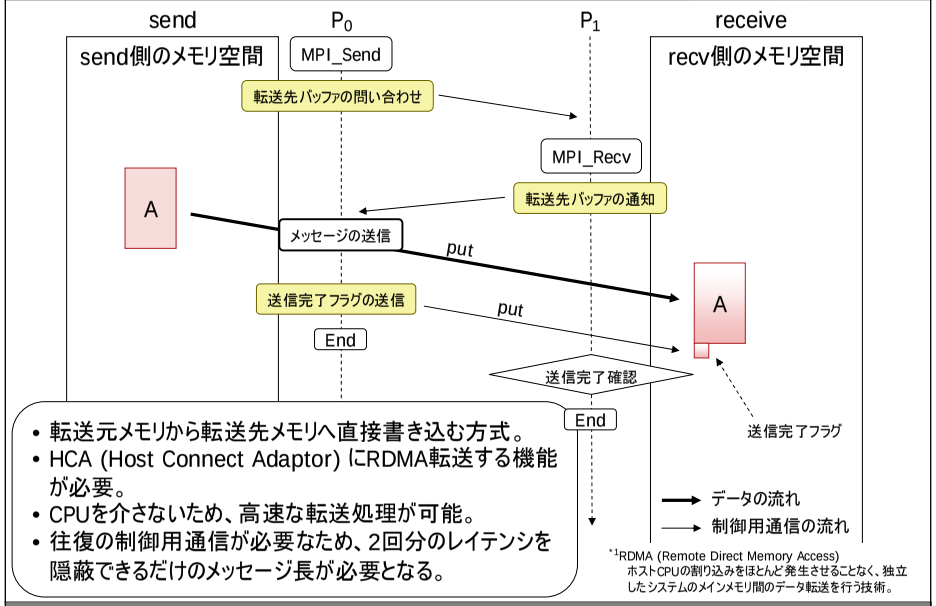
- 受信バッファを介した通信方式。
- recv側のバッファは固定なので、send側は受信バッファがある限り、いつでも送信することができる。
- 十分な受信バッファを必要とするため、メッセージ長が短い場合に使用される。

1対1通信プロトコル(rendezvous send/recv)



- 受信するための領域を、受信先へ問い合わせする方式。
- 送信側は、転送先に受信バッファを要求して、転送先が応えるまで転送しない。
- 最大で往復の制御用通信が必要なため、2回分のレイテンシを隠蔽できるだけのメッセージ長が必要となる。

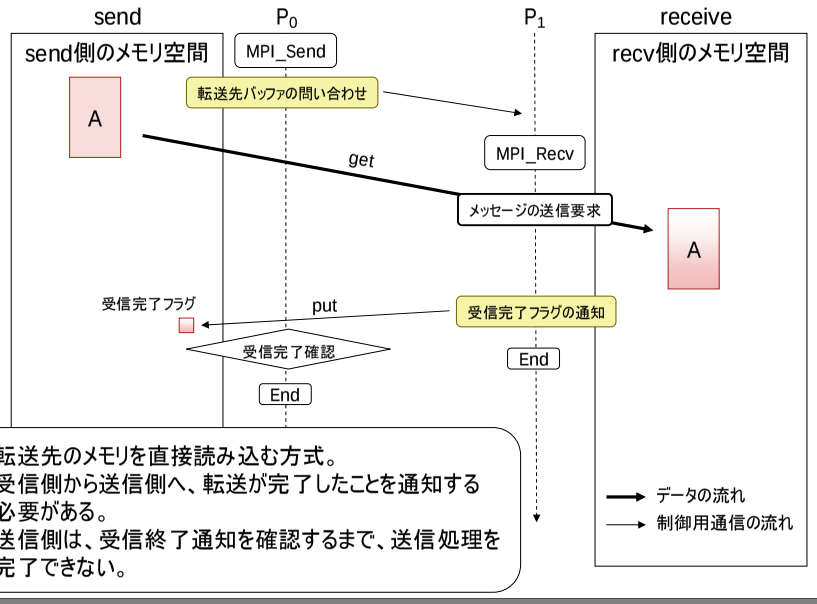
1対1通信プロトコル(RDMA*1 put)



- 転送元メモリから転送先メモリへ直接書き込む方式。
- HCA (Host Connect Adaptor) にRDMA転送する機能が必要。
- CPUを介さないため、高速な転送処理が可能。
- 往復の制御用通信が必要なため、2回分のレイテンシを隠蔽できるだけのメッセージ長が必要となる。

*1RDMA (Remote Direct Memory Access)
ホストCPUの割り込みをほとんど発生させることなく、独立したシステムのメインメモリ間のデータ転送を行う技術。

1対1通信プロトコル(RDMA get)



- 転送先のメモリを直接読み込む方式。
- 受信側から送信側へ、転送が完了したことを通知する必要がある。
- 送信側は、受信終了通知を確認するまで、送信処理を完了できない。

集団通信関数

- 集団通信とは
- 集団通信の機能
- 実装アルゴリズムの例
- Allgather
- Alltoall

集団通信とは

FUJITSU

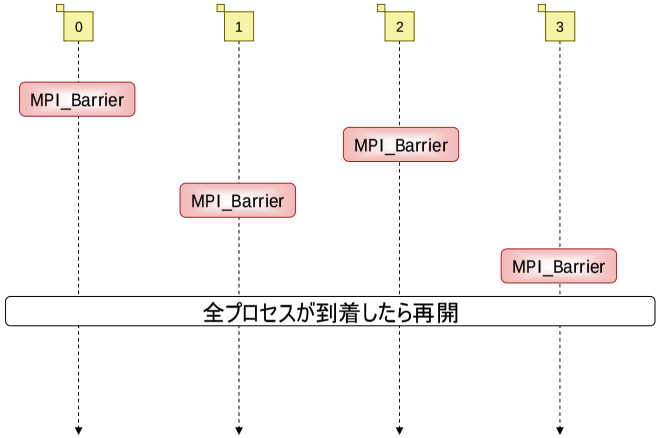
- あるプロセスグループ内でグループ内に関連した通信を定義しています。
- 全てのプロセスグループが、同じ引数を持って呼び出すことが前提となります。
- ブロッキング処理です。

MPIがサポートする集団通信関数

放送	MPI_Bcast
収集	MPI_Gather
	MPI_Gatherv
拡散	MPI_Scatter
	MPI_Scatterv
収集データを全メンバで共有	MPI_Allgather
	MPI_Allgatherv
完全交換	MPI_Alltoall
	MPI_Alltoallv
	MPI_Alltoallw
リダクション	MPI_Reduce
リダクション結果を全メンバで共有	MPI_Allreduce
リダクション結果を全メンバに拡散	MPI_Reduce_scatter
スキャン	MPI_Scan
バリア同期	MPI_Barrier

Barrier (バリア同期)

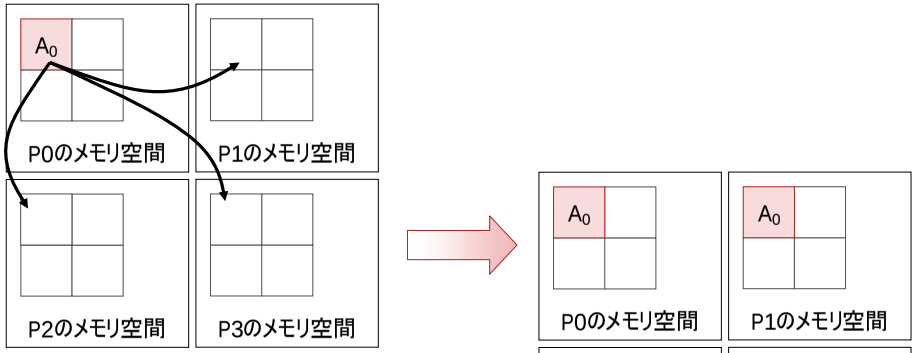
FUJITSU




```
MPI_Barrier(comm)

comm        コミュニケータ
```


Bcast (放送)

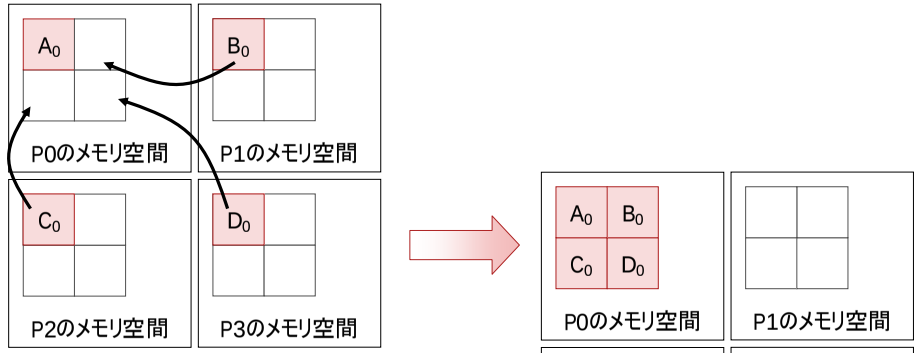
FUJITSU




```
MPI_Bcast(buffer, count, datatype, root, comm)

buffer      バッファの先頭アドレス
count       バッファ内の要素数
datatype    バッファのデータタイプ
root        ルートとなるランク
comm        コミュニケータ
```


Gather (収集)

FUJITSU

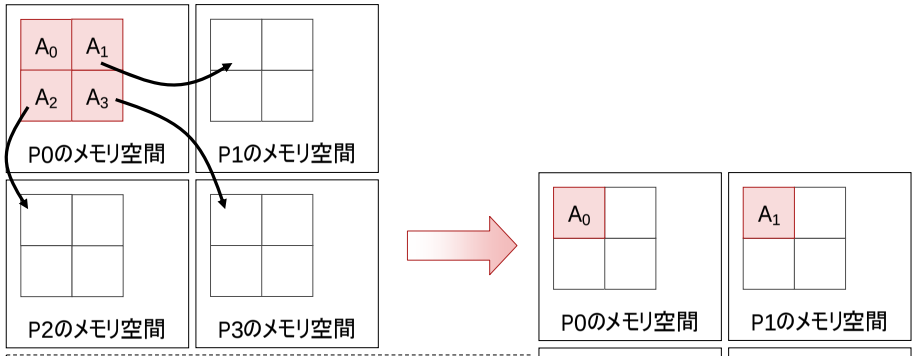



```
MPI_Gather(sendbuf, sendcount, sendtype, recvcount,
recvtype, root, comm)

sendbuf     送信バッファの先頭アドレス
sendcount   送信バッファの要素数
sendtype    送信バッファのデータタイプ
recvbuf     受信バッファの先頭アドレス(rootのみ有効)
recvcount   受信バッファの要素数(rootのみ有効)
recvtype    受信バッファのデータタイプ(rootのみ有効)
root        ルートとなるランク
comm        コミュニケータ
```

可変長のMPI_Gathervもある

Scatter (拡散=Gatherと反対の処理)

FUJITSU

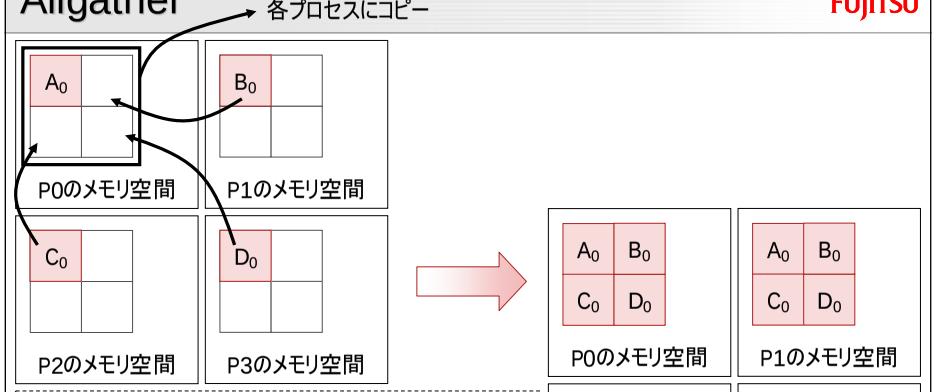



```
MPI_Scatter(sendbuf, sendcount, sendtype, recvcount,
recvtype, root, comm)

sendbuf    送信バッファの先頭アドレス(rootのみ有効)
sendcount  送信バッファの要素数(rootのみ有効)
sendtype   送信バッファのデータタイプ(rootのみ有効)
recvbuf    受信バッファの先頭アドレス
recvcount  受信バッファの要素数
recvtype   受信バッファのデータタイプ
root       ルートとなるランク
comm       コミュニケータ
```

可変長のMPI_Scattervもある

Allgather

FUJITSU

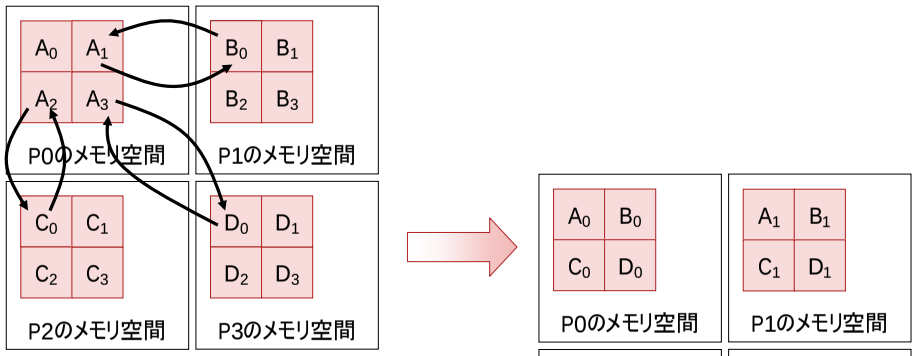



```
MPI_Allgather(sendbuf, sendcount, sendtype, recvcount,
recvtype,comm)

sendbuf    送信バッファの先頭アドレス
sendcount  送信バッファの要素数
sendtype   送信バッファのデータタイプ
recvbuf    受信バッファの先頭アドレス
recvcount  一プロセスから受信する要素数
recvtype   受信バッファのデータタイプ
comm       コミュニケータ
```

(参考) Gather + Bcastと同等の処理

Alltoall

FUJITSU

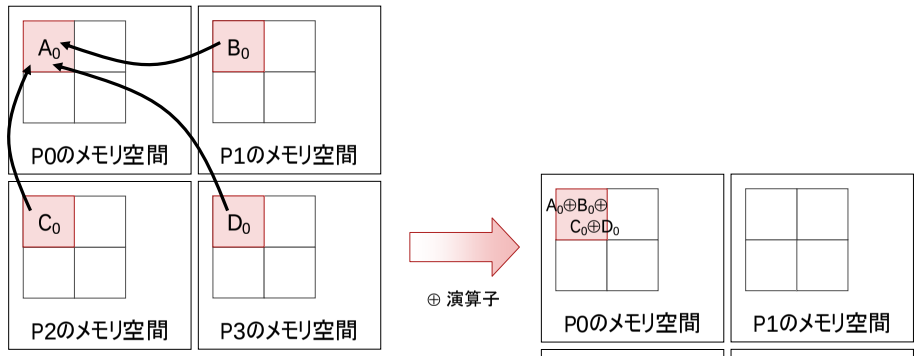



```
MPI_Alltoall(sendbuf, sendcount, sendtype, recvcount,
recvtype, comm)

sendbuf    送信バッファの先頭アドレス
sendcount  各プロセスへ送信する要素数
sendtype   送信バッファのデータタイプ
recvbuf    受信バッファの先頭アドレス
recvcount  一プロセスから受信する要素数
recvtype   受信バッファのデータタイプ
comm       コミュニケータ
```

可変長のMPI_AlltoallvやMPI_Alltoallwもある

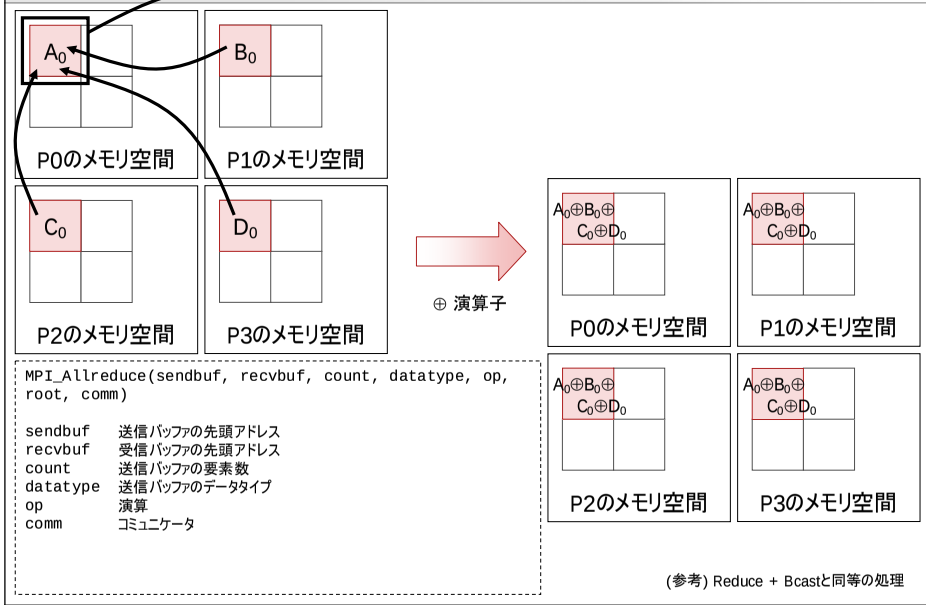
Reduce

FUJITSU




```
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root,
comm)

sendbuf    送信バッファの先頭アドレス
recvbuf    受信バッファの先頭アドレス
count      送信バッファの要素数
datatype   送信バッファのデータタイプ
op         演算
root       ルートとなるランク
comm       コミュニケータ
```

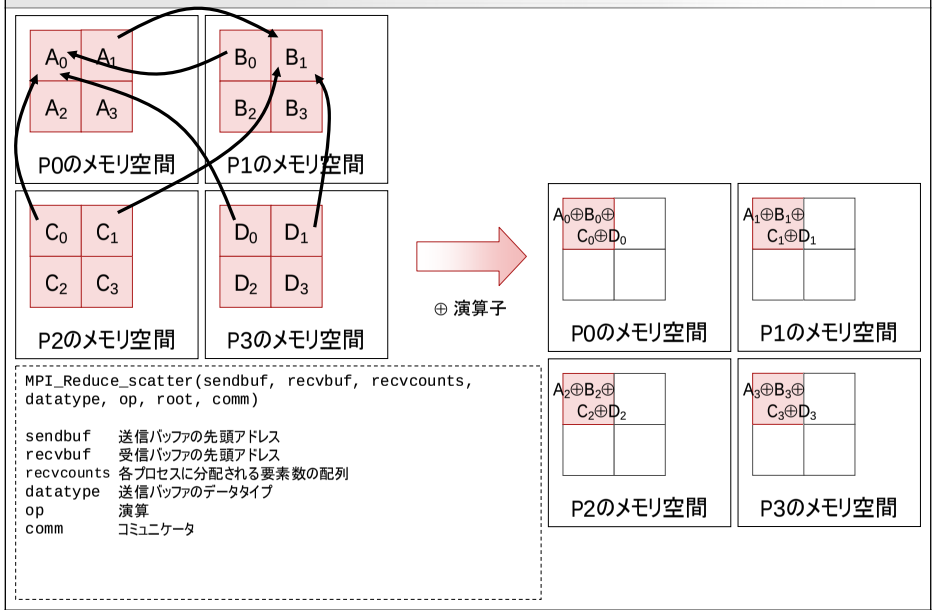
Allreduce




```
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op,
root, comm)

sendbuf   送信バッファの先頭アドレス
recvbuf   受信バッファの先頭アドレス
count     送信バッファの要素数
datatype  送信バッファのデータタイプ
op        演算
comm      コミュニケータ
```

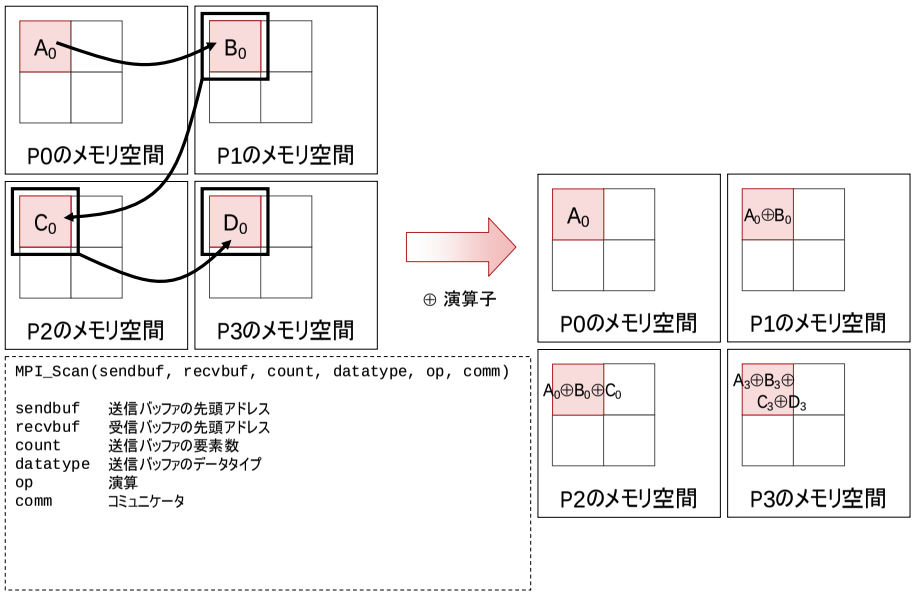
(参考) Reduce + Bcastと同等の処理

Reduce_scatter




```
MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts,
datatype, op, root, comm)

sendbuf   送信バッファの先頭アドレス
recvbuf   受信バッファの先頭アドレス
recvcounts 各プロセスに分配される要素数の配列
datatype  送信バッファのデータタイプ
op        演算
comm      コミュニケータ
```


Scan




```
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm)

sendbuf   送信バッファの先頭アドレス
recvbuf   受信バッファの先頭アドレス
count     送信バッファの要素数
datatype  送信バッファのデータタイプ
op        演算
comm      コミュニケータ
```


MPIが対応するリダクション演算

Op	Function	対応する型
MPI_MAX	最大値	C整数型・Fortran整数型,実数型,複素数型
MPI_MIN	最小値	C整数型・Fortran整数型,実数型,複素数型
MPI_SUM	和	C整数型・Fortran整数型,実数型,複素数型
MPI_PROD	積	C整数型・Fortran整数型,実数型,複素数型
MPI_LAND	論理積	C整数型,論理型
MPI_BAND	ビット演算の積	C整数型・Fortran整数型,バイト型
MPI_LOR	論理和	C整数型,論理型
MPI_BOR	ビット演算の和	C整数型・Fortran整数型,バイト型
MPI_LXOR	排他的論理和	C整数型,論理型
MPI_BXOR	ビット演算の排他的論理和	C整数型・Fortran整数型,バイト型
MPI_MAXLOC	最大値と位置	
MPI_MINLOC	最小値と位置	

集団通信のアルゴリズム例

- Allgatherのアルゴリズムの例
  - Ring Algorithm
  - Recursive Doubling Algorithm
  - Bruck Algorithm
  - Neighbor Exchange Algorithm
- Alltoallのアルゴリズムの例
  - Simple Spread Algorithm
  - Pair-wise Exchange Algorithm

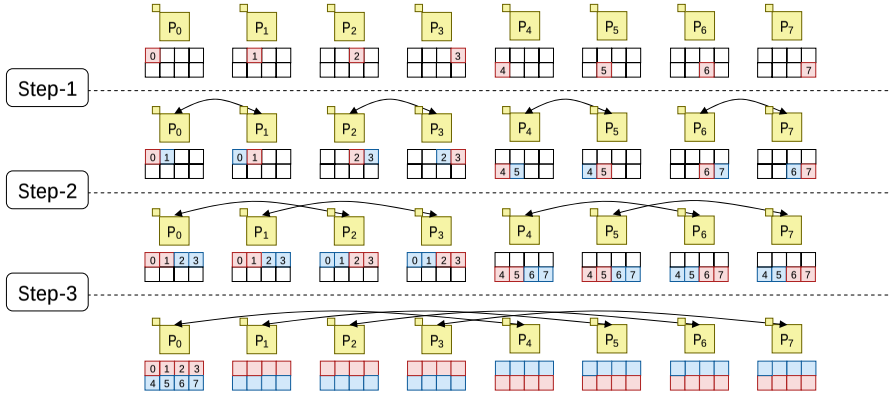
Algorithm <Allgather> (1/4)

Ring Algorithm

一プロセス分のデータを次々と隣に送る。ステップ数=(P-1)

Algorithm <Allgather> (2/4)

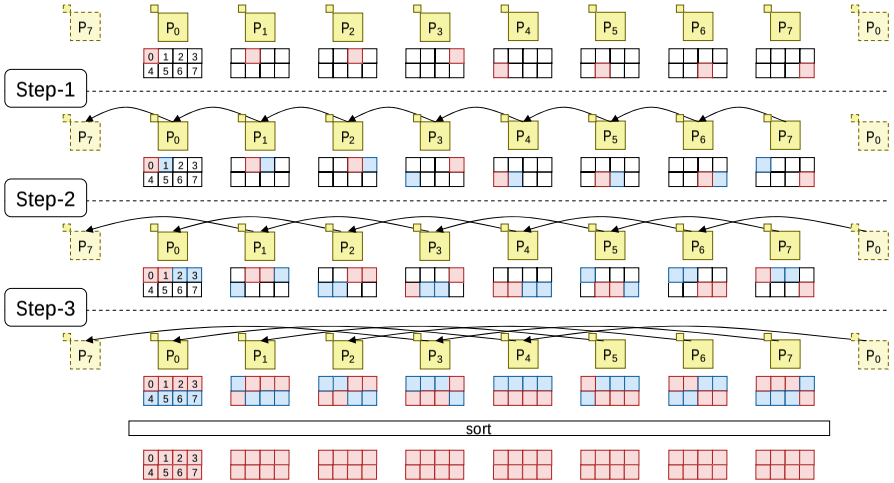
Recursive Doubling Algorithm



ステップ数(=log2P)が抑えられるが、2べきのプロセス数でないと適用できない。

Algorithm <Allgather> (3/4)

Bruck Algorithm

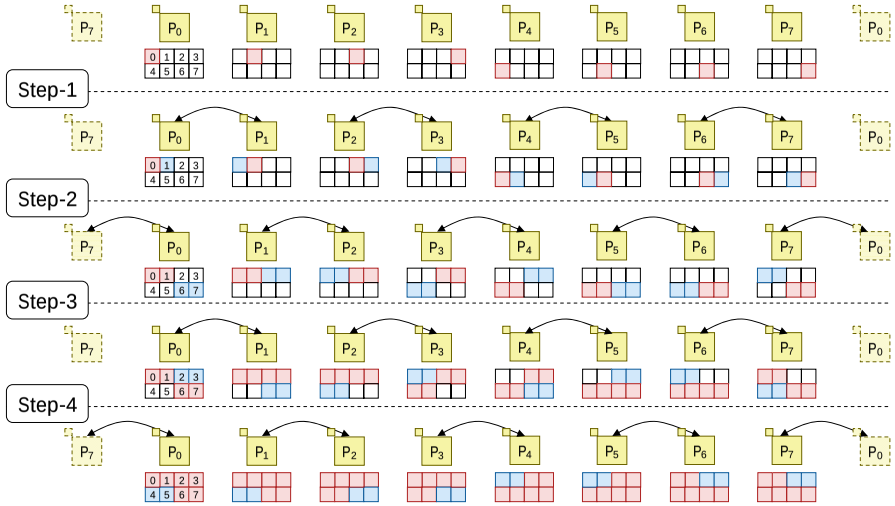


非連続データを扱うため、最後でソートが必要になる。ステップ数(=log2P)。

Algorithm <Allgather> (4/4)

FUJITSU

Neighbor Exchange Algorithm

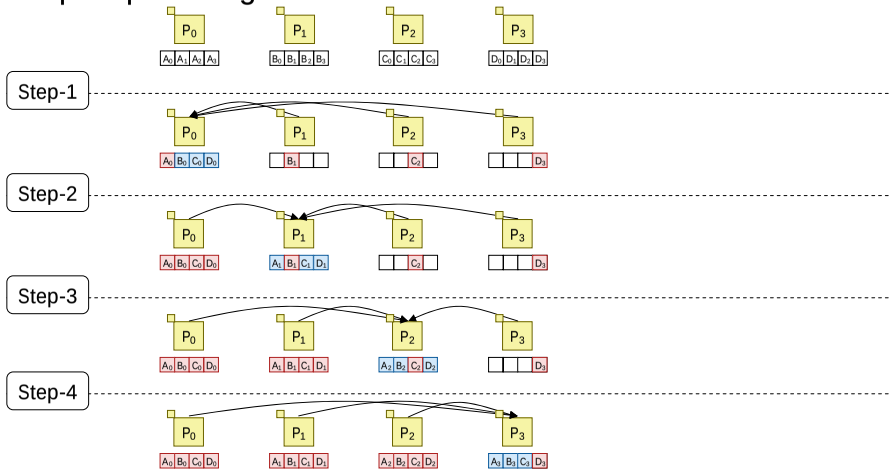


ステップ数(=P/2)はRecursive Doublingより多いが、通信は隣接通信のみ。

Algorithm <Alltoall> (1/2)

FUJITSU

Simple Spread Algorithm



一ステップで、一つのプロセスにすべてのデータを集める。ステップ数=P。

Algorithm <Alltoall> (2/2)

FUJITSU

Pair-wise Exchange Algorithm

一要素ずつ近くから遠くのランクへ転送する。ステップ数=P-1。

FUJITSU

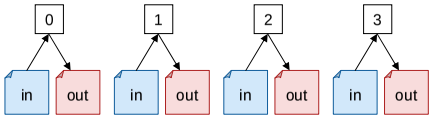
MPI-IO

- HPCにおけるIOアーキテクチャ
- アプリの状況
- Data Model (HDF5, NetCDF)
- MPI-IO規格
- ROMIO

HPCにおけるIO

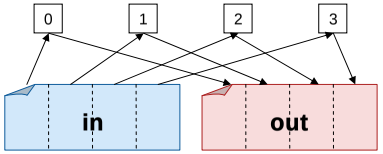
FUJITSU

- 現在のアプリケーションの傾向
  - 1プロセスごとに入力ファイル・出力ファイルが存在



特徴
- プロセス数が少ないうちは、高速にファイルへアクセスできるが、プロセス数が増える(作成するファイル数が増える)に連れて、ファイルシステムへの負荷が増大する。

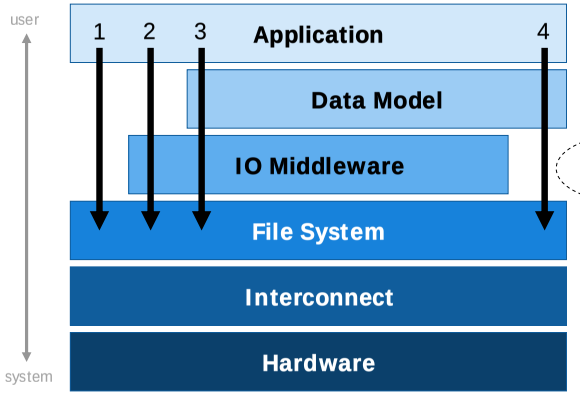
- 将来予測されるアプリケーションの傾向
  - 1アプリケーション全体で入力ファイルと出力ファイルを管理



特徴
- 分散並列上のアプリケーション全体から、一つのファイルへの入出力が可能なため、ファイルシステムへの負荷をかけずに入出力ができる。

HPCにおけるIOアーキテクチャ

FUJITSU



1. POSIX IOでの入出力
2. MPI-IOでの入出力
3. Data ModelがMPI-IOを利用して入出力
4. Data ModelがPOSIX IOを利用して入出力

MPI-IOとは、POSIXインターフェースでは実現できない、
- 複数プロセスが共有ファイルを効率的にアクセスするためのインターフェースを提供
- プロセス間IOのポータビリティ性を提供

している、MPI-2規格の一部です。

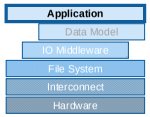
アプリの状況

FUJITSU

Science Domain	Code	Programming Language	Programming Model	I/O Libraries	Math Libraries
Accelerator Design	T3P	C/C++	MPI	NetCDF	MUMPS, ParMETIS, Zoltan
Astrophysics	CHIMERA	F90	MPI	HDF5, pNetCDF	LAPACK
Biology	LAMMPS	C/C++	MPI		FFTW
Chemistry	MADNESS	F90	MPI		BLAS
Chemistry	NWChem	F77, C/C++	MPI, Global Arrays, ARMCI		BLAS, ScaLAPACK, FFTPACK
Chemistry	ORelTran	F95	MPI		LAPACK
Climate	CAM	F90, C, CAF	MPI, OpenMP	NetCDF	SciLib
Climate	POP/CICE	F90, CAF	MPI, OpenMP	NetCDF	
Climate	MITgcm	F90, C	MPI, OpenMP	NetCDF	
Combustion	S3D	F90	MPI		
Fusion	AORSA	F77, F90		NetCDF	ScaLAPACK, FFTPACK
Fusion	GTC	F90, C/C++	MPI, OpenMP	MPI-IO, HDF5, NetCDF, XML	PetSC
Fusion	GYRO	F90, Python	MPI	MPI-IO, NetCDF	BLAS, LAPACK, UMPACK, MUMPS, FFTW, SciLib, ESSL
Geophysics	PFLOTRAN	F90	MPI		BLAS, PetSC
Materials Science	LSMS	F77, F90, C/C++	MPI2	HDF5, XML	BLAS, LAPACK
Materials Science	QBOX	C/C++	MPI	XML	LAPACK, ScaLAPACK, FFTW
Materials Science	QMC	F90	MPI		BLAS, LAPACK, SPRNG
Nanoscience	CASINO	F90	MPI		BLAS
Nanoscience	VASP	F90	MPI		BLAS, ScaLAPACK
Nuclear Energy	NEWTRNX	F90, C/C++, Python		HDF5	LAPACK, PARPACK
Nuclear Physics	CCSD	F90	MPI	MPI-IO	BLAS
QCD	MILC, Chroma	C/C++	MPI		

NCCS High Performance Computing Allocations for FY 2006
Dr. Bronson Messer (Oak Ridge National Laboratory)

- アメリカでの主要アプリの状況は?
  - まだまだ、Fortranが主力
  - 高レベルIOは、HDF5とNetCDFが主流
  - MPI-IOで記述されたアプリが3本存在
  - Parallel NetCDFは間接的にMPI-IOを使用しています。
  - HDF5は間接的にMPI-IOを使用する可能性があります。
  - POSIX IOのジョブもまだまだ多い。

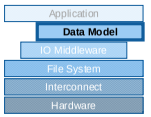


データモデルの役割

FUJITSU

- 多様なAPIの提供
  - C/C++/Fortranに加えて、最近のトレンドはJavaをサポートしています
- ランダムアクセス
  - ランダムアクセスの提供により、目的のデータを高速に取り出すことができます。
- 既存データの読み出し
  - データモデルがサポートする処理系ならば、ファイルを読み書きすることができます。
  - ファイルの転送先やリモートアクセスでも、ファイルを読み書きすることができます。

  ⇒ データファイルの一般提供が可能
- ファイルの圧縮
  - zlibによるデータ圧縮をサポートしています。
- 並列IOによるアクセス
  - より高速化するために、並列IOのサポートが開始されています。(Parallel NetCDFとHDF5)



HDF5の特徴

FUJITSU

- 特徴
  - NCSA(アメリカ国立スーパーコンピュータ応用研究所)で開発された、階層データフォーマット
  - 複雑なデータオブジェクトや大量のメタデータを表現できるデータモデル
  - データオブジェクトの数や大きさに制限のない可搬性の高いファイルフォーマット
  - C, C++, Fortran 90, JavaにAPIを提供
  - Virtual File Layerにより、用途に応じたファイルを出力

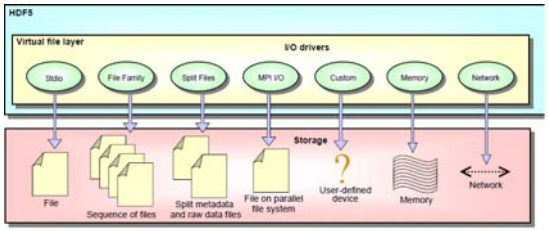


Figure 10.3: The HDF5 virtual file layer (VFL). Through the use of the VFL, an HDF5 file can be stored as a conventional UNIX file or as multiple files (to overcome a 32-bit system limit on file size); it can be stored as two or more files containing separated meta data and raw data; it can be stored as multiple files on a parallel file system; it can be saved in memory or sent over the network; or it can be handled by another non-standard, user-supplied driver.

HDF5 Wins 2002 R&D 100 Award
http://hdf.ncsa.uiuc.edu/HDF5/RD100-2002/All_About_HDF5.pdf

NetCDF・Parallel NetCDFの特徴

FUJITSU

- 特徴
  - UCAR (University Corporation for Atmospheric Research)の、Unidata Centerが開発したデータモデル
  - Network-transparent / Self-Describing
    - NetCDFファイルは、どの環境(たとえばエンディアンが違っても)であっても、ユーザーは変換することなく、データを取り出すことができます。
    - NetCDFファイルをネットワーク転送したり、リモートアクセスすることも可能です。
  - C, C++, FOTRAN77, Fortran90, JavaにAPIを提供
  - 第三者によって、MATLAB, Objective-C, Perl, Python, R, Ruby, Tcl/Tkで利用できるAPIを提供
  - Parallel NetCDFは、NetCDFを拡張して、MPI-IOを使用して並列化
    ⇒内部的に並列IOを使用するため、ユーザーが意識することなく、並列ファイルシステムを有効に活用できます。
  - 主に気象系データの解析に使用

MPI-IOの概要

FUJITSU

- 集団的操作によるファイルのオープンとクローズ

```
MPI_FILE_OPEN(comm, filename, amode, info, fh)
IN      comm        コミュニケータ
IN      filename    ファイル名
IN      amode       アクセスモード
IN      info        infoオブジェクト
OUT     fh          ファイルハンドル
```

効率的にアクセスするための情報を、ヒントとして処理系に渡すことができます

```
MPI_FILE_CLOSE(fh)
INOUT   fh          ファイルハンドル
```

MPI-IOは、プロセス間で共有する一つのファイルにアクセスすることができます。(プロセスごとに別のファイルを作成することも可能です)

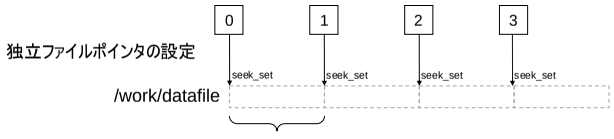
MPI-IOの動作イメージ

FUJITSU

一つのファイルに複数のプロセスから書き込みを行うことができます

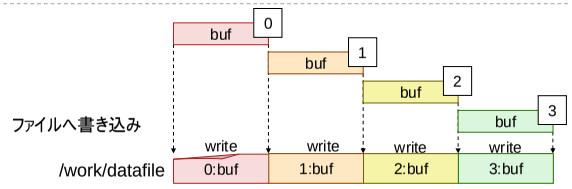
```
MPI_File_open(MPI_COMM_WORLD, "/work/datafile",
    MPI_MODE_CREATE | MPI_MODE_WRONLY,
    MPI_INFO_NULL, &fh);
MPI_File_seek(fh, rank*bufsize, MPI_SEEK_SET);
```

ファイルをオープンして、オフセットを設定する




```
MPI_File_write(fh, buf, elements, MPI_INT, &s);
```

MPI_INT型をelements個書き込む




```
MPI_File_close(&fh);
```

ファイルシーク(明示的オフセット)

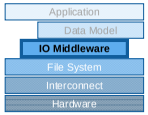
FUJITSU

- 3種類のファイルシークを提供 (1/3)
  - ファイルの先頭を0としたときのオフセット値を指定して、ファイルへアクセスします。ファイルポインタは更新されません。ファイルのオープン時に、MPI_MODE_SEQUENTIALモードを指定した場合は、利用できません。

```
MPI_FILE_WRITE_AT(fh, offset, buf, count,
datatype, status)
INOUT     fh                  ファイルハンドル
IN        offset              オフセット
IN        buf                 バッファの先頭アドレス
IN        count               バッファの要素数
IN        datatype            バッファのデータタイプ
OUT       status              ステータスオブジェクト
```

```
MPI_FILE_READ_AT(fh, offset, buf, count, datatype,
status)
IN        fh                  ファイルハンドル
IN        offset              オフセット
OUT       buf                 バッファの先頭アドレス
IN        count               バッファの要素数
IN        datatype            バッファのデータタイプ
OUT       status              ステータスオブジェクト
```

```
MPI_FILE_READ_AT          関連API
MPI_FILE_WRITE_AT
MPI_FILE_IREAD_AT
MPI_FILE_IWRITE_AT
MPI_FILE_READ_AT_ALL
MPI_FILE_WRITE_AT_ALL
MPI_FILE_READ_AT_ALL_BEGIN
MPI_FILE_READ_AT_ALL_END
MPI_FILE_WRITE_AT_ALL_BEGIN
MPI_FILE_WRITE_AT_ALL_END
```



ファイルシーク(独立ファイルポインタ)

FUJITSU

- 3種類のファイルシークを提供 (2/3)
  - 各プロセスが独立したファイルポインタを持って、ファイルへアクセスします。共有ファイルポインタは更新されません。ファイルのオープン時に、MPI_MODE_SEQUENTIALモードを指定した場合は、利用できません。

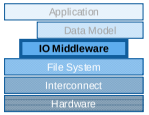
```
MPI_FILE_WRITE(fh, buf, count, datatype, status)
INOUT     fh                  ファイルハンドル
IN        buf                 バッファの先頭アドレス
IN        count               バッファの要素数
IN        datatype            バッファのデータタイプ
OUT       status              ステータスオブジェクト
```

```
MPI_FILE_READ(fh, buf, count, datatype, status)
INOUT     fh                  ファイルハンドル
OUT       buf                 バッファの先頭アドレス
IN        count               バッファの要素数
IN        datatype            バッファのデータタイプ
OUT       status              ステータスオブジェクト
```

```
MPI_FILE_SEEK(fh, offset, whence)
INOUT     fh                  ファイルハンドル
IN        offset              オフセット
IN        whence              更新モード
```

```
MPI_FILE_GET_POSITION(fh, offset)
IN        fh                  ファイルハンドル
OUT       offset              オフセット
```

```
MPI_FILE_READ             関連API
MPI_FILE_WRITE
MPI_FILE_IREAD
MPI_FILE_IWRITE
MPI_FILE_READ_ALL
MPI_FILE_WRITE_ALL
MPI_FILE_READ_ALL_BEGIN
MPI_FILE_READ_ALL_END
MPI_FILE_WRITE_ALL_BEGIN
MPI_FILE_WRITE_ALL_END
```



ファイルシーク(共有ファイルポインタ)

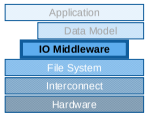
FUJITSU

- 3種類のファイルシークを提供 (3/3)
  - 各プロセス間で共通のファイルポインタを持って、ファイルへアクセスします。同時に共有ファイルポインタを使用したデータアクセスをする場合、シリアライズされて実行されますが、その順序は処理系依存です。順序保証するには、ユーザー自身で同期処理を行う必要があります。

```
MPI_FILE_WRITE_SHARED(fh, buf, count, datatype,
status)
INOUT     fh                  ファイルハンドル
IN        offset              オフセット
IN        count               バッファの要素数
IN        datatype            バッファのデータタイプ
OUT       status              ステータスオブジェクト
```

```
MPI_FILE_READ_SHARED(fh, buf, count, datatype,
status)
IN        fh                  ファイルハンドル
OUT       buf                 バッファの先頭アドレス
IN        count               バッファの要素数
IN        datatype            バッファのデータタイプ
OUT       status              ステータスオブジェクト
```

```
MPI_FILE_READ_SHARED      関連API
MPI_FILE_WRITE_SHARED
MPI_FILE_IREAD_SHARED
MPI_FILE_IWRITE_SHARED
MPI_FILE_READ_ORDERED
MPI_FILE_WRITE_ORDERED
MPI_FILE_READ_ORDERED_BEGIN
MPI_FILE_READ_ORDERED_END
MPI_FILE_WRITE_ORDERED_BEGIN
MPI_FILE_WRITE_ORDERED_END
```



ブロッキングとノンブロッキング

FUJITSU

- ブロッキングIOとノンブロッキングIO

  1対1通信と同様に、ブロッキングIOとノンブロッキングIOが提供されています。

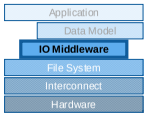
  - ブロッキングIO

    IO要求が完了するまで戻りません。ただし、書き込みバッファを利用するシステムでは、記憶デバイスへの書き込みを保証するものではありません。記憶デバイスへの転送を保証するためには、MPI_FILE_SYNCを使用します。

```
MPI_FILE_SYNC(fh)
INOUT     fh                  ファイルハンドル
```

  - ノンブロッキングIO

    IO操作の開始ルーチンを呼び出しますが、完了するまで待ちません。実際のIO処理を行っている間に、他の演算処理をオーバーラップすることができます。ユーザーバッファが再利用可能かどうかは、ノンブロッキング通信と同様に、MPI_WaitやMPI_Testなどを使用して確認します。



集団的データアクセス

FUJITSU

- 非集団的データアクセスと集団的データアクセス

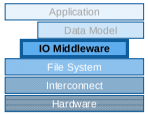
  1対1通信と同様に、非集団的IOと集団的IOが提供されています。

  - 非集団的データアクセス

    処理の完了は、呼び出したプロセスのみに依存します。

  - 集団的データアクセス

    処理の完了は、集団的呼び出しに参加する全てのプロセスに依存します。しかし、大域的データアクセスの場合、MPIライブラリ内に入出力の最適化の余地が十分にある場合は、性能面で有利になる可能性があります。



ファイル情報

FUJITSU

- ファイル情報

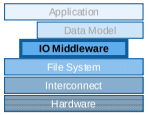
  ユーザーは、最適化の指示として、infoを通じてシステム側にヒントを渡すことができます。

```
MPI_FILE_SET_INFO(fh, info)
INOUT     fh                ファイルハンドル
IN        info              infoオブジェクト
```

```
MPI_FILE_GET_INFO(fh, info_used)
INOUT     fh                ファイルハンドル
OUT       info_used         infoオブジェクト
```

(参考) Fujitsu MPI(V3系)がサポートするinfoオブジェクトのkey

key	value(省略値)	意味
cb_buffer_size	4194304	集団的アクセスに使用する一時バッファの大きさ
cb_nodes	コミュニケータの大きさ	集団的アクセスで実際に入出力を行うプロセス数
ind_rd_buffer_size	4194304	プロセス個別読み込み時のバッファ域の大きさ
ind_wr_buffer_size	524288	プロセス個別書き込み時のバッファ域の大きさ



MPI-IOにおけるデータ表現(1)

FUJITSU

- 3種類のデータ表現
  - MPI-IOは、データファイルのポータビリティのために、3つのデータ表現を用意しています。
  - “native”

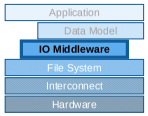
    データ表現は、メモリ中のバイナリイメージと同じです。同一機種内でデータファイルを使用する場合は、精度が保証され、変換コストが省略できる利点があります。
  - “internal”

    ファイル形式は、同一システム内(異機種間であっても)で保証されます。データ表現は、MPIの実装によって決められますが、”external32”として置き換えられる場合があります。
  - “external32”

    データ表現は、他システムのMPIでも保証されます。データ変換が必要な処理系では、精度が失われたり、実行性能の低下を招きます。”external32”は、MPI-2規格で厳密に定義されていますが、一般的なIEEE754+ビッグエンディアン形式(SPARCのデータフォーマット)がイメージされています。

    - 全ての浮動小数点型はIEEE型のビッグエンディアンで表現される。4倍精度は、16バイト(指数部15ビット、仮数部112ビット)とする。
    - 全ての整数型は、2の補数のビッグエンディアンで表現される。
    - FortranのLOGICAL型とC++のbool型は、偽を0、真を非零とする。
    - FortranのCOMPLEX型とDOUBLE COMPLEX型は、二つの浮動小数点フォーマットの組み合わせとする。
    - 文字型はISO 8859-1とし、MPI_WCHARはUnicodeフォーマットとする。
    - NaNは伝搬するものとする。

(注意) Fujitsu MPI(V3系)は、”internal”と”external32”は、”native”で解釈されます。



MPI-IOにおけるデータ表現(2)

FUJITSU

- 3種類のデータ表現の設定と参照

```
MPI_FILE_SET_VIEW(fh, disp, etype, filetype,
datarep, info)
INOUT     fh                ファイルハンドル
IN        disp              オフセット
IN        etype             基本データ型
IN        filetype          ファイル型
IN        datarep           データ表現
IN        info              infoオブジェクト
```

独立ファイルポインタ・共有ファイルポインタは更新される

“native”, “internal”, “external32”のいずれかを指定する

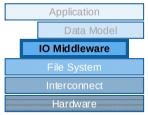
```
MPI_FILE_GET_VIEW(fh, disp, etype, filetype,
datarep)
IN        fh                ファイルハンドル
OUT       disp              オフセット
OUT       etype             基本データ型
OUT       filetype          ファイル型
OUT       datarep           データ表現
```

MPIの実装(ROMIO)

FUJITSU

- ROMIO
  - Argonne National Laboratoryによって実装されたMPI-IO
  - MPI-IO実装のde facto standard
    - MPICH
    - LAM-MPI
    - HP MPI
    - SGI MPI
    - NEC MPI
    - 富士通MPI(V3系から)

    } などに採用済み
  - adioレイヤー構成により、ファイルシステムの追加が容易
  - 対応する豊富なファイルシステム
    - IBM PIOFS
    - Intel PFS
    - HP/Convex HFS
    - SGI XFS
    - NEC SFS
    - PVFS
    - NFS
    - UFS



FUJITSU

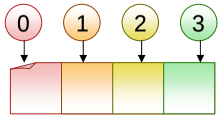
MPIでのファイル出力

- MPI-IO方式
- Split Files方式
- IOマスター方式

MPI-IO方式によるファイル転送

FUJITSU

- MPI-IOを利用して一つのファイルに転送(MPI-IO方式)



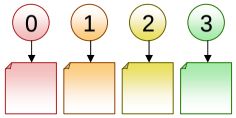

```
MPI_FILE_DELETE(fname);

MPI_File_open(MPI_COMM_WORLD, filename,
    MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI_File_seek(fh, myrank * bufsize, MPI_SEEK_SET);
MPI_File_write(fh, buf, elements, MPI_DOUBLE, &status);
MPI_File_sync(fh);
MPI_File_close(&fh);
```

- ユーザーはプロセスの同期を取ることなく、一つのファイルを作成することができる。
- 多重書き込みをサポートする高速なネットワーク共有ファイルシステムが前提となる。

Split Files方式によるファイル転送

FUJITSU

- プロセスごとにファイルを転送(Split Files方式)



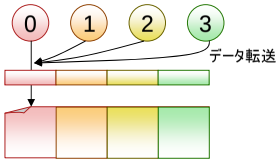

```
sprintf(filename, "%s.%03d", BASENAME, this_rank);
unlink(filename);

fd = open(filename, O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR);
write(fd, (void *)buf, bufsize);
fsync(fd);
close(fd);
```

- 一般的なPOSIX-IOで実現できるため、わかりやすいプログラムで記述できる。
- ファイルがプロセス数個できるため、プロセス増加に伴い、ファイルシステムに負荷がかかる。

IOマスター方式によるファイル転送

FUJITSU

- ルートに出力データを集めて、一つのファイルを転送(IOマスター方式)




```
unlink(filename);

MPI_Gather(buf, (int)elements, MPI_DOUBLE,
    allbuf, (int)elements, MPI_DOUBLE, 0, MPI_COMM_WORLD);
if(this_rank == 0){
    fd = open(filename, O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR);
    write(fd, (void *)allbuf, allbufsize);
    fsync(fd);
    close(fd);
}
```

- 事前にデータ転送が必要なため、通信コストが高い。
- ルートプロセスの負荷が高く、プロセスの負荷バランスが悪くなる。

FUJITSU

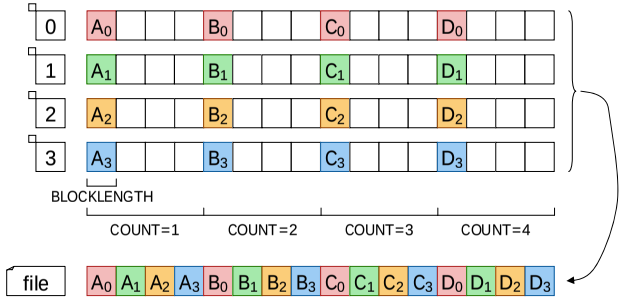
MPI-IOでのストライド転送

- ストライドデータの入出力
- Data Sieving

ストライドデータの入出力

FUJITSU

- 非連続ベクトル型データから一つのファイルを作成。



ベクトル宣言型

```
size = MPI_Comm_size(MPI_COMM_WORLD, &size);

MPI_Type_contiguous(BLOCKLENGTH, MPI_INT, &type1);
MPI_Type_commit(&type1);

MPI_Type_vector(COUNT, BLOCKLENGTH, BLOCKLENGTH * size, MPI_INT, &type2);
MPI_Type_commit(&type2);
```


POSIX-IOとMPI-IOでの違い

FUJITSU

- POSIX-IOでの書き込み

```
y = malloc(BLOCKLENGTH * size * COUNT * sizeof(int));
for(j = 0; j < COUNT; j++)
    MPI_Gather(x + size * j * BLOCKLENGTH, 1, type1, y + size * j * BLOCKLENGTH,
               1, type1, 0, MPI_COMM_WORLD);
if(rank == 0){
    fd = open(fname, O_CREAT | O_WRONLY, 0666);
    write(fd, y, BLOCKLENGTH * size * COUNT * sizeof(int));
    fsync(fd);
    close(fd);
}
free(y);
```

POSIX-IOは、IOマスター方式でなければ書き込めない

- MPI-IOでの書き込み

```
MPI_File_open(MPI_COMM_WORLD, (char *)fname,
              MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, BLOCKLENGTH * rank * sizeof(int), type1, type2,
                  "native", MPI_INFO_NULL);
MPI_File_write_all(fh, x, 1, type2, &status);
MPI_File_sync(fh);
MPI_File_close(&fh);
```

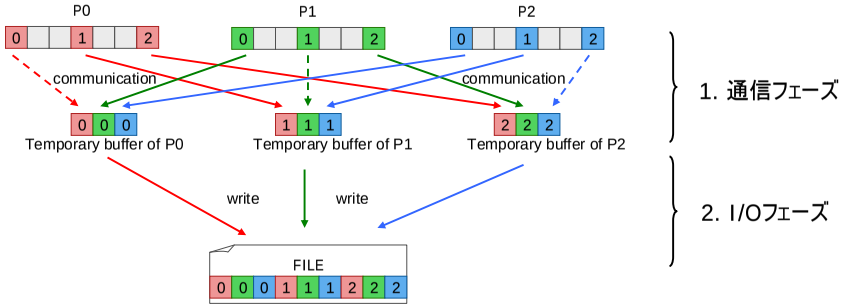
MPI-IOは、型を意識した入出力が可能

さらに、集団的MPI-IOを用いると、
Data Sievingを用いた効率的な入出力を行うことができる

Data Sievingとは

FUJITSU

- ROMIOの集団的MPI-IOに実装されたデータバッファリング
  - 不連続データの集団的MPI-IOで使用されます。
  - 一つ一つのブロックに対する小さいファイル要求でなく、バッファを使用した大きな領域でのファイル要求となるため、性能面で有利に働くことがあります。
  - 通信フェーズとI/Oフェーズの2つのフェーズがあります。writeの場合は、通信フェーズ→I/Oフェーズとなり、readの場合は、I/Oフェーズ→通信フェーズとなります。
  - Data Sievingを利用するかどうかとテンポラリバッファの大きさは、ヒントにより変更可能です。
    - romio_ds_read / romio_ds_write　　Data Sievingを利用するかどうか
    - ind_rd_buffer_size / ind_wr_buffer_size　　テンポラリバッファの大きさ



3.3.2. MPI-IO の実アプリへの適用

ー3 次元構造格子のベクトルデータプログラム MPI-IO 化ー

富士通株式会社　　杉崎 由典

1. はじめに

MPI で並列化されたプログラムの入出力処理は、多くの場合で split file 処理方式と呼ばれる処理方式を用いている。Split file 処理方式は、プロセス毎に 1 つのファイルを生成する処理方式である。そのため、プロセス数が増えると生成されるファイル数もその分多くなる。

HPC においては、並列数は 100 や 1000 を超えることも珍しくないが、並列数分のファイルを入出力処理で扱う必要が生じると、ファイル管理が困難になることや、ファイルシステムへの負荷が増大する恐れがある。

一方、並列 IO 処理である MPI-IO 機能を用いることによって、複数プロセス実行であっても生成されるファイル数を 1 つに抑えることが可能である。

そこで、この MPI-IO 機能を用い、生成されるファイル数を 1 つに抑えることを実アプリで検証した。実アプリは、3 次元構造格子のベクトルデータプログラムを用いた。

2. ファイル転送方式

split file 方式と MPI-IO 方式の違いを提示する。

2.1. split file 方式

プロセス毎にファイルを生成する split file 方式について説明する。

以下の図の様に、split file 方式は、各プロセスがそれぞれファイルを生成する。（〇はプロセス、□はファイルを表す）

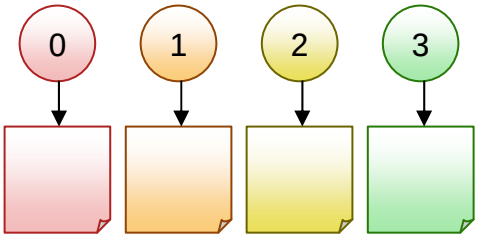


図1. split file 方式概略

```
sprintf(filename, “%s.%03d", BASENAME, this_rank);
unlink(filename);

fd = open(filename, O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR);
write(fd, (void *)buf, bufsize);
fsync(fd);
close(fd);
```

図2. プログラム　Split file 方式

以下に split file 方式の特徴を挙げる。

・一般的な POSIX-IO で実現できるため、わかりやすいプログラムで記述できる。

・ファイルがプロセス数個できるため、プロセス増加に伴い、ファイルシステムに負荷がかかる。

2.2. MPI-IO 方式

複数のプロセスがあってもファイルは 1 つだけを生成する MPI-IO 方式について説明する。

以下の図の様に、MPI-IO 方式は、各プロセスがそれぞれ 1 つのファイルを分割して、各プロセスに割り当てられた部分に対して転送を行う。生成されたファイルは 1 つとして見える。(○はプロセス、□はファイルを現す)

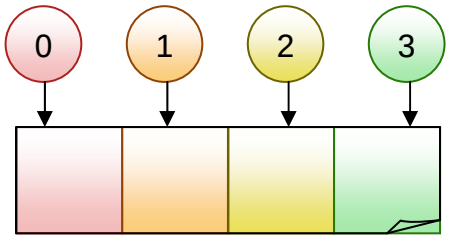


図3. MPI-IO 方式概略

```
MPI_FILE_DELETE(fname);

MPI_File_open(MPI_COMM_WORLD, filename,
    MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI_File_seek(fh, myrank * bufsize, MPI_SEEK_SET);
MPI_File_write(fh, buf, elements, MPI_DOUBLE, &status);
MPI_File_sync(fh);
MPI_File_close(&fh);
```

図4. プログラム　MPI-IO 方式

以下に MPI-IO 方式の特徴を挙げる。

・ユーザーはプロセスの同期を取ることなく、一つのファイルを作成することができる。
・多重書き込みをサポートする高速なネットワーク共有ファイルシステムが前提となる。

3. 検証プログラム

今回検証対象としたのは 3 次元構造格子のベクトルデータプログラムである。

3.1. プログラム構造

このプログラムのプログラム構造と IO 処理部の特徴を以下に示す。

・プログラム構造
- 8 並列プログラム
- プロセスを 2×2×2 の 3 次元に分割(2×2×2 のカルテシアン空間)
- IO は各プロセス毎に実施
- IO 処理は OUTP,OUTR で実施

・IO 処理
- write 文(DO 型並び)による配列の出力
- 配列のサイズは、各プロセスで異なる。

例：配列 U の場合　宣言サイズ U(63,63,243)
rank=0 U(60,60,210,3)
rank=1 U(60,60,211,3)
rank=2 U(60,61,210,3)
rank=3 U(60,61,211,3)
rank=4 U(61,60,210,3)
rank=5 U(61,60,211,3)
rank=6 U(61,61,210,3)
rank=7 U(61,61,211,3)

3.2. MPI-IO 対象ファイル

3 次元カルテシアン空間に分けられ、それぞれの次元を 000 または 001 で現すことで、下記の 5 ファイルが対象となる。

MPI-IO 対象ファイル

fort_xxx_yyy_zzz.20	xxx は 000 または 001,yyy は 000 または 001,zzz は 000 または 001
fort_xxx_yyy_zzz.21	
fort_xxx_yyy_zzz.22	
fort_xxx_yyy_zzz.23	
fort_xxx_yyy_zzz.50	

3.2. MPI-IO 化イメージ

元のプログラムで生成されるファイルと MPI-IO 化後のファイルのイメージを以下に示す。

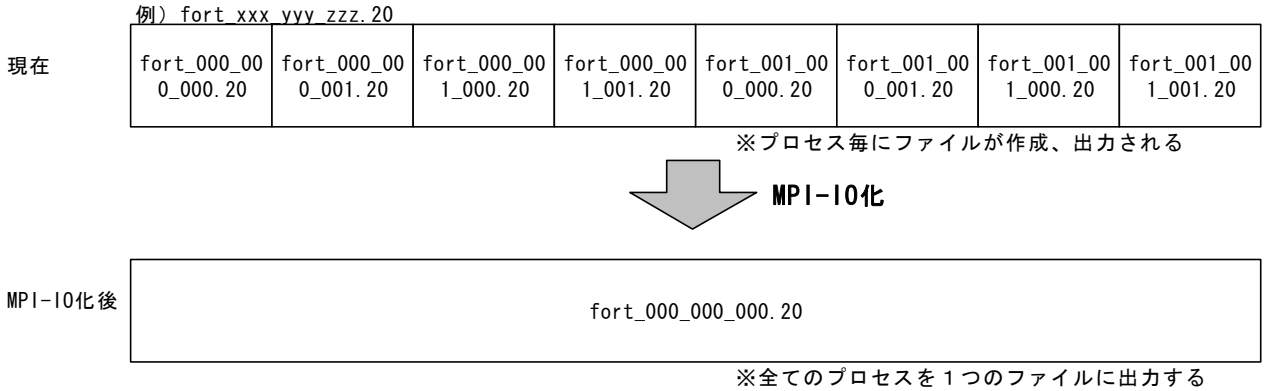


図5. MPI-IO 化イメージ

4. MPI-IO 適用方法

今回実施した MPI-IO 化の方法を示す。

4.1. IO 部の変形

以下の 2 つの方法がある。

・MPI-IO 化前に IO 部を変形

- 複数 write 文をまとめて以下の様に変換

```
write(20) (((U(I1,I2,I3,1),I1=1,MYI1MAX),I2=1,MYI2MAX),I3=1,MYI3MAX)
write(20) (((U(I1,I2,I3,2),I1=1,MYI1MAX),I2=1,MYI2MAX),I3=1,MYI3MAX)
write(20) (((U(I1,I2,I3,3),I1=1,MYI1MAX),I2=1,MYI2MAX),I3=1,MYI3MAX)
    ↓
write(20) U
```

・IO 部変換しない場合

- 各プロセスのインデックス 3 次元分を MPI_allgather で各プロセスに分配
- 上記 3 次元分のインデックスから、自 rank の offset 値を算出
  ただし、この方法は offset 計算が煩雑になる。上記の配列 U のケースに適用するためには、U(…,1),U(…,2),U(…,3)それぞれで offset 計算しなければならない。

上記理由から、IO 部をまとめて write する方法を採用した。

4.2. MPI-IO 化

元ソースから MPI-IO 化するためのポイントと、その変更方法を以下に示す。例として write のケースを挙げた。

以下の手順で MPI-IO 化を行った。

- OPEN 文、CLOSE 文、WRITE 文をそれぞれ、MPI_FILE_OPEN、MPI_FILE_CLOSE、MPI_FILE_WRITE_AT に変換する。
- MPI_FILE_WRITE_AT の引数にある、ファイル先頭からのオフセット値の計算処理を追加する。

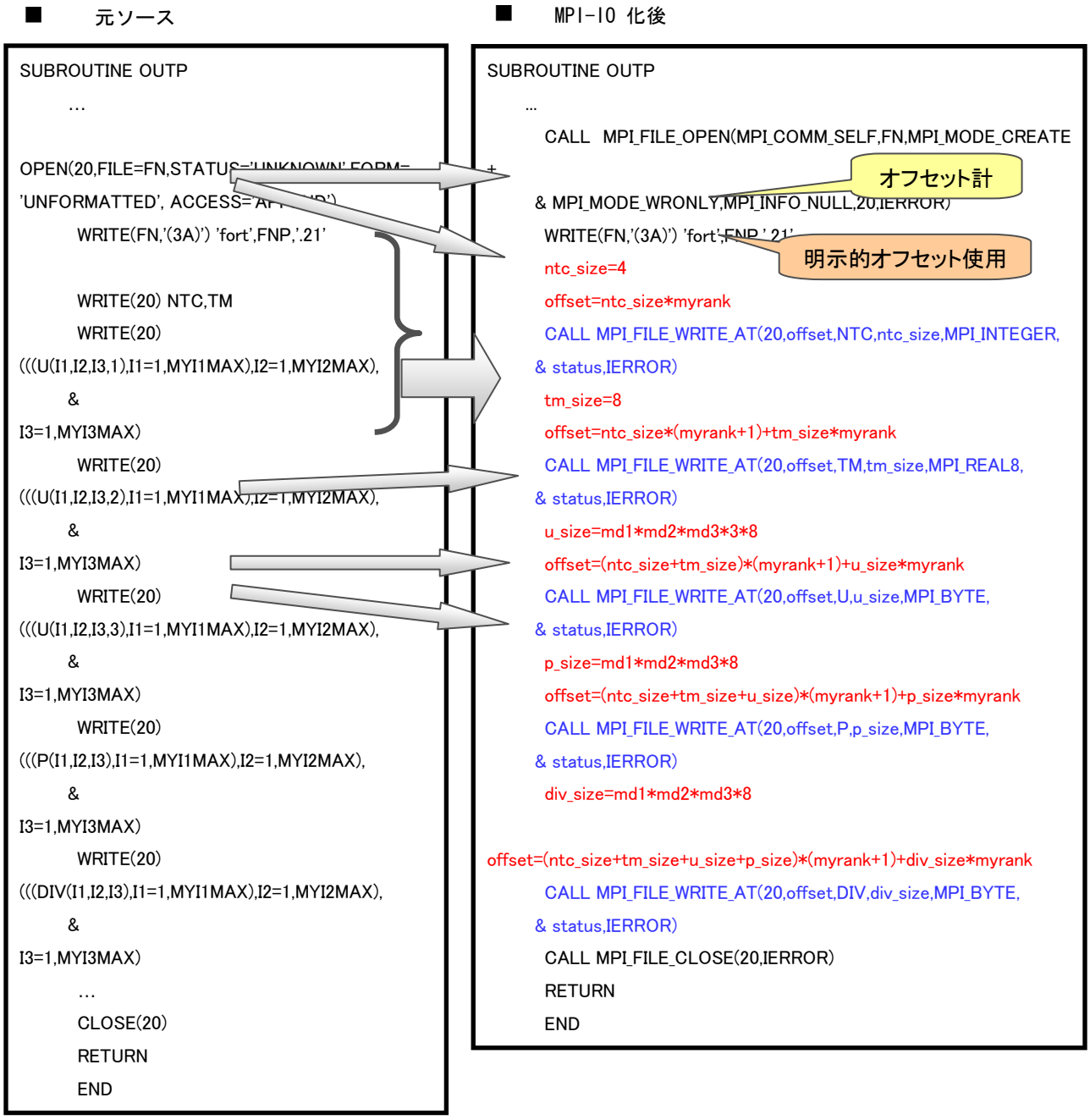


図6. MPI-IO 化実施例

4.3. 出力ファイル

MPI-IO 化前と MPI-IO 化後の出力ファイルのサイズを以下に挙げる。例としてファイル装置番号 20 番のケースを挙げた。

・MPI-IO 化前(write(20))

サイズ(Byte)	ファイル名
30240060	fort_000_000_000.20
30384060	fort_000_000_001.20
30744060	fort_000_001_000.20
30890460	fort_000_001_001.20
30744060	fort_001_000_000.20
30890460	fort_001_000_001.20
31256460	fort_001_001_000.20
31405300	fort_001_001_001.20
246554920	合計

・MPI-IO 化後(write(20))
　　サイズ(Byte)　ファイル名
　　277766496　fort_000_000_000.20

5. 実施状況

MPI-IO 化の実施状況及び結果を以下に示す。

・一般ユーザーが書き換えすることを想定
- Fortran プログラミングの経験があり、MPI の経験がない者が実施。
- MPI-IO 化の手順のみを提示。（本稿の 2.1 split file 方式,2.2 MPI-IO 方式の説明レベル）

・期間
- 約 10 日間で実施。（一日 4 時間）
  プログラムの解析で約 3 日間。
  MPI-IO 化とデバッグまでを約 7 日間。（ファイルが出力されるまで）

・状況
- 長時間ジョブとなるため、演算部の回数を縮小して実施。
  500step を 1 ステップで実施。IO 部には影響なし。
- プログラミング(MPI-IO への書き換え)は順当に実施できた。
  手順が判れば比較的簡単。
- MPI_FILE_SYNC はなし。
- IO 処理が集約されているプログラムであったため、修正箇所を限定して作業できた。

6. 評価

実施結果から MPI-IO 化のポイントについて抽出した。以下に示す。

・MPI 化の移行工数
- 変換する MPI-IO の関数が判れば、通常の作業工数で書き換え可能。
- 予想よりは難しくない感触。

・MPI-IO 化のポイント
- 書き換え方法の明確化（指針）
- IO 対象を全配列化（全配列対象に書き換え）
- オフセット計算を間違えない（注意・対策・仕組み）

・注意点／課題
- オフセット計算が間違い易い
  対策：デバッグを考慮しながら作業。(デバッグ出力を挿入)
- デバッグ方法の検討

本稿では、1 つのアプリを題材として MPI-IO 化を実施し、MPI-IO 化の手順や MPI-IO 化のポイントを抽出した。MPI-IO 化のポイントを押さえれば、それ程難しくなく通常の作業工数で書き換えが可能であることが判った。ただし、デバッグ方法は検討課題である。

7. 参考文献

- Message Passing Interface Forum　http://www.mpi-forum.org/

以上

THE POSSIBILITIES ARE INFINITE FUJITSU

3.4. プログラミング指針

ーフラットMPIとハイブリッド並列についてー

富士通株式会社　青木 正樹、鈴木 清文

目次

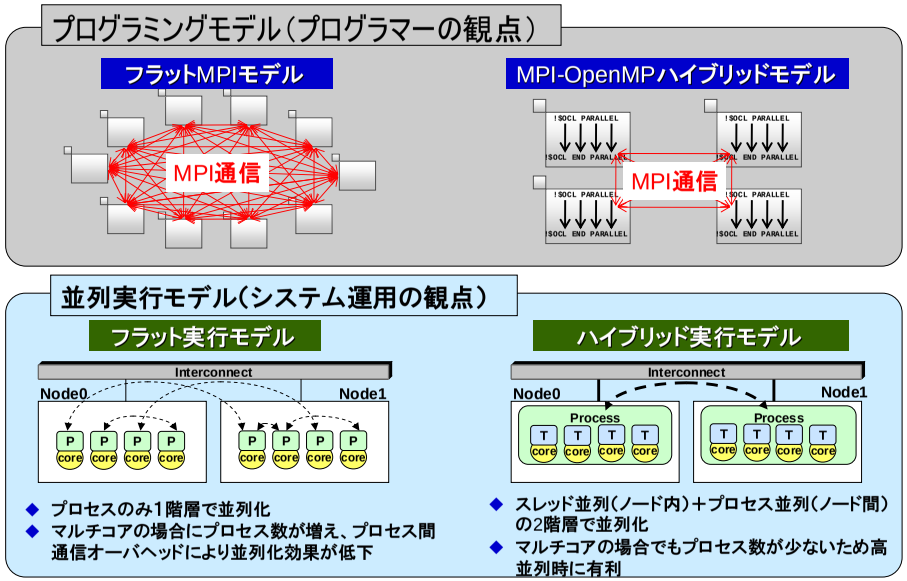
参考資料

- 【補足資料】高機能スイッチの位置づけと効果
  - 「効果例」で使用されている高機能スイッチ（高機能SWあるいはISWと記述）についての解説です。
- SMPクラスタWG成果報告書
  「スカラ並列プログラミング チューニング・ガイド」
  http://www.ssken.gr.jp/MAINSITE/activity/workinggroup/smpc/guide.html
  - スカラSMP計算機(富士通PRIMEPOWERシリーズ)上でプログラムの性能評価作業およびチューニングを行う際のガイドです。『SMPクラスタWG』の成果として作成されました。(2003/12/25発行)

プログラミングモデル

プログラミングモデルと並列実行モデル

FUJITSU



- プロセスのみ1階層で並列化
- マルチコアの場合にプロセス数が増え、プロセス間通信オーバヘッドにより並列化効果が低下
- スレッド並列（ノード内）＋プロセス並列（ノード間）の2階層で並列化
- マルチコアの場合でもプロセス数が少ないため高並列時に有利

並列プログラミング環境

FUJITSU

プログラミングモデル/プロセス並列手法		ノード間	ノード内
ハイブリッドモデル	データ並列	XPFortran	OpenMP, 自動並列（スレッド並列）
	メッセージパッシング	MPI	
フラットモデル	データ並列	XPFortran	
	メッセージパッシング	MPI	

プログラミングモデル選択のポイント

FUJITSU

モデル	メリット	デメリット
ハイブリッド並列（＋SIMD化）	プロセス数増加を抑える	スレッド並列性能に依存（OpenMP化も考慮）
フラットMPI（＋SIMD化）	1階層のみの考慮でよい	プロセス数増加の弊害(通信オーバーヘッド増大)

結局のところ、

フラットMPIの性能特性　VS　スレッド並列の”プログラミング”×”性能”

がプログラミングモデル選択のポイントとなる。

FUJITSU

VISIMPACT
（性能とプログラミングの容易性）

VISIMPACTとは(1)

FUJITSU

(VISIMPACT:
VIrtual Single processor by Integrated Multi-core Parallel ArChiTecture)

●コンセプト
- マルチコアCPU内の高効率なスレッド並列を実現する技術
- 高効率なハイブリッド並列実行モデルの実現を支援

MPI + スレッド並列処理(自動並列 /OpenMP)

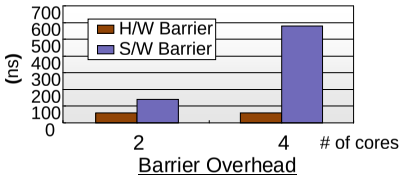
●狙い
- マルチコアCPUを高速な一つのCPUを扱うことで……
  - MPI プロセス数を $1/n_{core}$ に減らす
    - ➔並列処理効率を向上
  - メモリアクセスを軽減

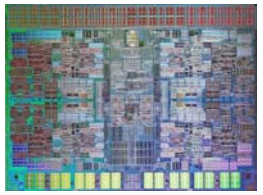
●技術的挑戦
- コア間のスレッドレベル並列処理のオーバヘッドをどう減らすか?

VISIMPACTとは(2)

FUJITSU

■CPU 技術
- コア間のハードバリア機能
  - ➔ソフトウェアバリアと比較して10倍の高速化
  - ➔コア数に依存せずに一定のオーバヘッドを実現





SPARC64™ VII
Real quad-core CPU for Technical Computing
(2.5 GHz, 40 GFlops/chip)

- 共有L2キャッシュメモリ (6 MB)
  - ➔キャッシュ間のメモリ交換を軽減
  - ➔キャッシュメモリの利用を効率化

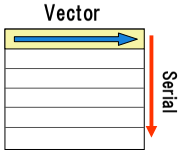
●コンパイラ技術
- ベクトル化の技術を適用した、高効率なスレッド並列(自動並列、OpenMP) を実現

VISIMPACTループ処理の考え方

FUJITSU

■Vector処理

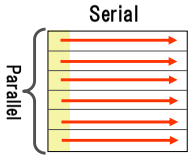
```
V DO J=1,N
V   DO I=1,M
V     A(I,J)=A(I,J+1)*B(I,J)
V   END
    END
```



- 適用範囲は広い
- 同期は頻繁だが低コスト

●従来の並列処理
(疎粒度並列)

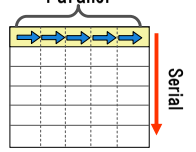
```
P   DO J=1,N
P     DO I=1,M
P       A(I,J)=A(I,J)*B(I,J)
P     END
P   END
```



- 適用範囲は狭い(広範囲な分析が必要)
- 同期は少ない

●今回の並列処理
(細粒度並列)

```
P   DO J=1,N
P     DO I=1,M
P       A(I,J)=A(I,J+1)*B(I,J)
P     END
    END
```



- 適用範囲は広い
- 同期は頻繁
- キャシュ間のデータの取り合いが頻繁に発生

VISIMPACT が対処

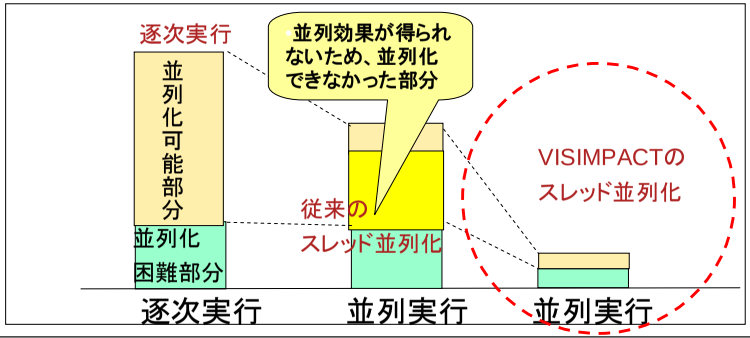
VISIMPACTの効果

FUJITSU

従来のスレッド並列化の問題
プロセス並列やベクトル化に比べ並列オーバヘッドが大きい。
そのため細粒度並列では並列効果が低くスレッド並列が利用しずらかった。

⇩⇩ ⇩⇩VISIMPACTにより解決⇩⇩⇩⇩

並列処理のオーバヘッド(同期、フォルスシュア)削減の結果、コンパイラによる細粒度並列化が可能となった。

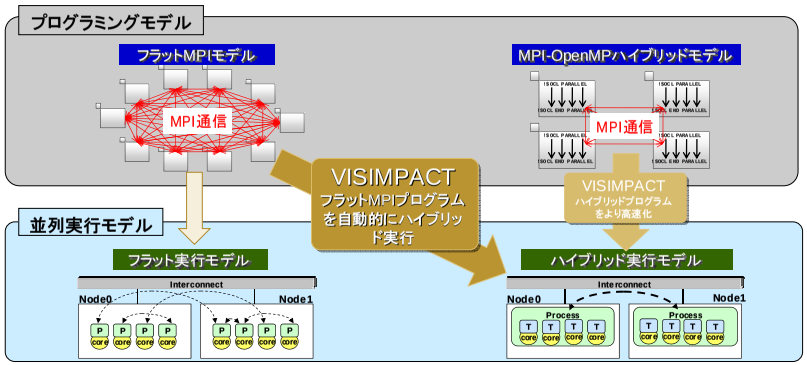


容易なハイブリッド実行の実現

FUJITSU

◆VISIMPACT
高効率なCPUチップ内(コア間)自動スレッド並列で、複数コアの仮想単一CPU化を実現

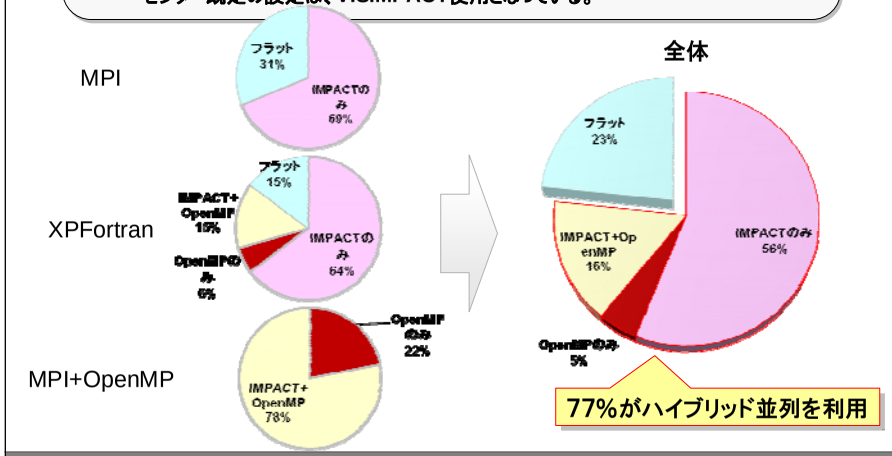
◆VISIMPACTによるフラットMPIプログラムのハイブリッド化



参考:JAXAセンターでのハイブリッド並列の利用状況

FUJITSU

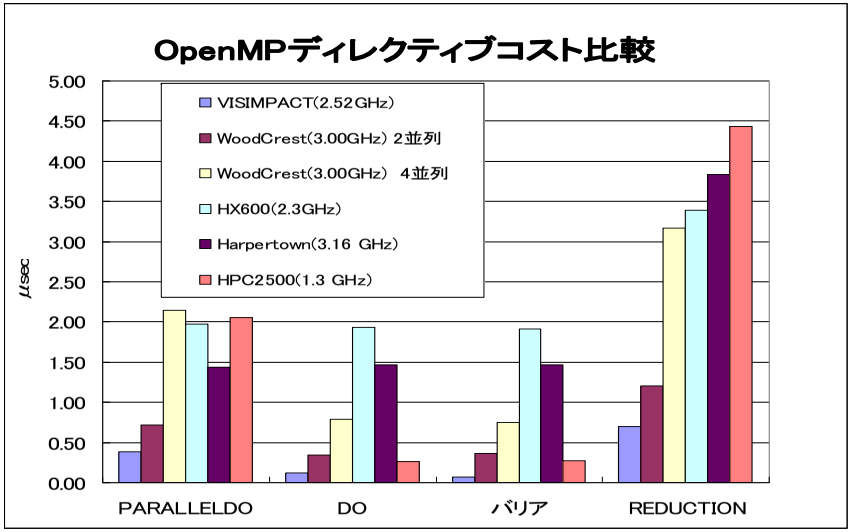
- コンパイル時ログの情報から利用状況を判断
  - 情報採取期間:2009.1～2009.9、コンパイル実施者数:145人
  - 情報数に重複あり(1人の利用者が異なるスタイルでコンパイルした場合)
  - センター既定の設定は、VISIMPACT使用となっている。



検証:スレッド並列のオーバーヘッド

FUJITSU

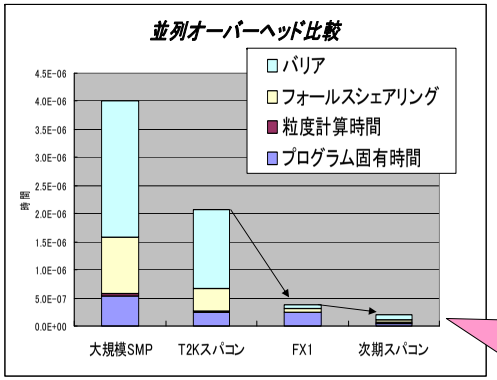
- スレッド並列のオーバーヘッドでFX1は他システムを圧倒



検証:並列化のオーバヘッド実測と推定

FUJITSU

- 大規模SMP(HPC2500),T2Kスパコン(HX600),FX1(SPARC64Ⅶ)で並列化のオーバヘッドの詳細を実測(4スレッド)。



検証用プログラム(n=900)

```
t0 = wclock()
!$omp parallel private(j,jsw)
      Do j = 1, nrep
        jsw = jswitch( krit, indj, j )
!$omp do
        Do i = 1, n
          y(i,jsw) = y(i,jsw) + c0*x1(i)
        End Do
!$omp end do
      End Do
!$omp end parallel
      t1 = wclock() - t0
```

これからのアプリ開発者は、スレッドへ並列のオーバヘッドを気にする必要なし!

スレッド並列化プログラミングの容易性

FUJITSU

高い並列化能力(*)を持つ自動並列化コンパイラ（＋OpenMP指示行）がプログラミングを支援。

■ 従来自動並列化
（疎粒度並列）
外側ループで
並列化を行ってきた。

■ VISIMPACTの自動並列化
（細粒度/疎粒度並列）
細粒度でも並列化効果あるため内側ループ(*)でも外側ループでも並列化可能。

＊ベクトル化と同様に内側ループを並列化（細粒度）
→技術的に確立しているベクトル化並みの
高い自動並列化率が可能

自動並列化能力

FUJITSU

ハードウェアのサポート（VISIMPACT）により、ベクトル化より広い範囲に適用！

		ベクトル化	VISIMPACTスレッド並列（計画）	現行スレッド並列
範囲		○（最内ループ）	◎（最内ループ＋手続き）	△（外側ループ）
データ型		○（4/8バイト型）	◎（全て）	◎（全て）
データ依存関係	依存なし	○	○	○
	順方向依存	○	○	×
	逆方向依存	×	○	×
ループの演算内容	四則演算	○	○	○
	リダクション演算	○	○	○
	収集・拡散	○	○（計画）	×
	DOブランチ	○	○（計画）	×
粒度		○（ループ長数＋）	○（ループ長数＋）	×（ループ長数千）

ANLベクトル化コンテストプログラム（全135ループ）を用いて、各種ループの解析能力を比較

	ベクトル化	VISIMPACTレッド並列（計画）	現行スレッド並列
ベクトル化、並列化可能なループ数	86ループ	89ループ以上	67ループ

スレッド並列化支援：スレッド並列化の確認情報

FUJITSU

言語開発環境において以下を支援。

■ 実行時情報：スレッド並列化率

■ ソースコード情報：並列化（Pソース）表示

表示例　PPマーク:並列化対象のループ　Pマーク:並列化対象の文

```
(line-no.)(nest)(optimize)

   7      1      p              do i=2,n
   8      2    pp 8               do j=2,n
   9      2     p 8                 a(i,j)=a(i,j)+b(i,j)*c
  10      2     p 8               enddo
  11      1      p              enddo
```

備考：最適化情報として「SIMD化」「ループ交換」「一重化」「prefetch情報」等もあり

スレッド並列化支援：OpenMP指示行の自動生成機能

FUJITSU

翻訳オプションに**-Qm**を指定することで、コンパイルリストに加え、自動並列化の状況をOpenMP指示行により表現した原始プログラムを生成します。（**プラットフォーム無依存コードを開発するための支援機能**）

```
frt -Kparallel -Qm sample.f90
        ↓
sample.omp.f90（生成されるソースファイル）
```

以下に、生成されるソースの例を示します。

例：オリジナルソース (sample.f90)

```
parameter      (imax=124, jmax=124)
real      *4   array( imax, jmax)

    do 10  j = 1, jmax
    do 10  i = 1, imax
      array(i,j) = array(i,j) + (i*j)
10  enddo
    end
```

例：生成されるソース（sample.omp.f90）

```
parameter      (imax=124, jmax=124)
real      *4   array( imax, jmax)

!$OMP PARALLEL DO PRIVATE(i) !FRT
    do 10  j = 1, jmax
    do 10  i = 1, imax
      array(i,j) = array(i,j) + (i*j)
10  enddo
    end
```

スレッド並列化（まとめ）：容易に高性能プログラミングが可能

FUJITSU

- プログラミングの容易性

  自動ベクトル化　≒　VISIMPACTの自動並列化

  ベクトル化プログラミングは容易：万人が認知。

  なぜ、ベクトル化プログラミングは容易なのか？

  ①細粒度のループを対象

  ②対象ループは、ループ繰り返しによるデータ依存性なし

  VISIMPACTの自動並列化の基本ﾌﾟﾛｸﾞﾗﾐﾝｸﾞもﾍﾞｸﾄﾙ化に類似

- 演算性能向上への相乗効果

  コア内の演算器を効率良く使うためには、ILP(*)を最大限に高める必要がある。

  ILP：命令レベルの並列性(Instruction-level parallelism)

  利用者は、細粒度の並列性を意識すること（上記②）により

  スカラCPUの演算性能を最大限引き出すことが可能

  →ループ繰り返しによるデータ依存なくなり

  ループ中の命令スケジューリングやSIMD化が促進

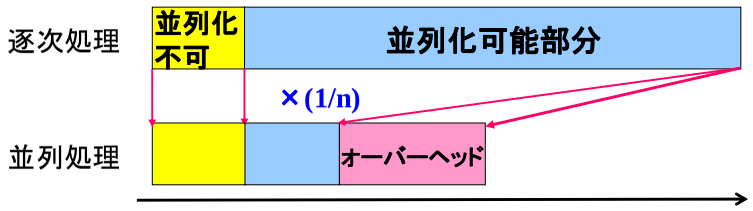
FUJITSU

VISIMPACTプログラミングの指針
（自動並列化プログラミング）

自動並列化プログラミングのポイント

FUJITSU

- 並列化率の向上
- 大きな並列化粒度
- メモリアクセス競合の削減
- CPU負荷の分散



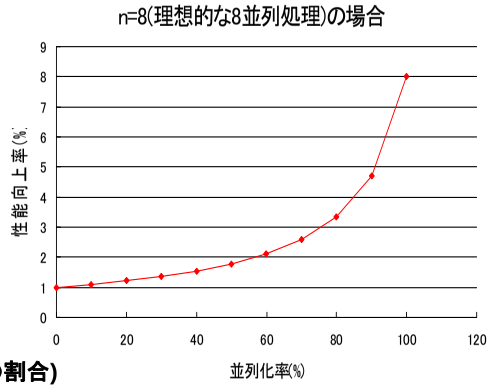
並列化率の向上

FUJITSU

- アムダールの法則

プログラム全体の性能向上率

$$= \frac{1}{(1-p)+\frac{p}{n}}$$



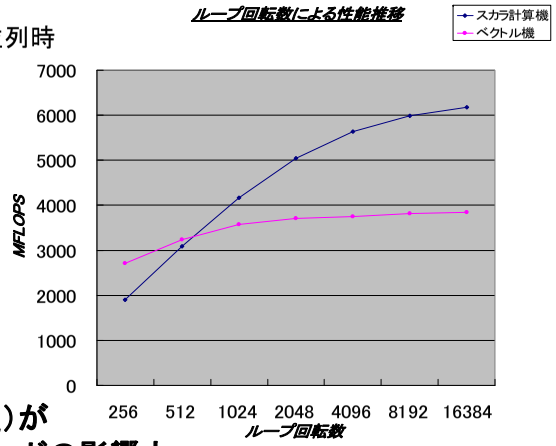
- p: 並列化率(並列化可能部分の割合)
- n: 並列化された部分の速度向上率

⇒並列化率の向上が重要！

並列化の粒度を大きく

FUJITSU

8並列時




```
DO  I=1,N
 A(I)=EXP(B(I))
ENDDO
```

評価ループソース

EXP関数を25演算と仮定

一般に、ループ長（粒度）が
小さいと並列化オーバヘッドの影響大

⇒粒度を大きく（外側で並列化）

⇒ただし、VISIMPACTでは影響小

メモリアクセス競合の削減

FUJITSU

■メモリアクセス（キャッシュ）競合の発生例と改善策

```
real(8)
    a(256,256),b(256,256),
    c(256,256),d(256,256),
    e(256,256),f(256,256)

do j=1,256
  do i=1,256
    a(i,j)=b(i,j)+c(i,j)*d(i,j)
          +e(i,j)*f(i,j)
  enddo
enddo
```

⇒

```
real(8)
    a(257,256),b(257,256),
    c(257,256),d(257,256),
    e(257,256),f(257,256)

do j=1,256
  do i=1,256
    a(i,j)=b(i,j)+c(i,j)*d(i,j)
          +e(i,j)*f(i,j)
  enddo
enddo
```

キャッシュ機構の特性から2のべき乗のデータサイズを持つ大きなデータ間でキャッシュの競合が発生しやすい。

改善策：

配列の1次元目の大きさ変更（＋1）

（または、-Karraypad系オプション指定）

CPU負荷の分散

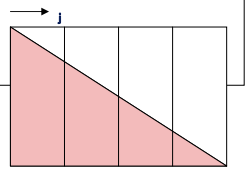
FUJITSU

■CPU負荷の分散

特定のCPUの負荷（実行時間）が大きいと、全体の実行時間がそのCPUの実行時間にひきづられる。

⇒各CPUの負荷（分担）は均等になるのが望ましい

```
【三角ループ】不均等例1
DO J=1,M  ←均等分割で並列化
DO I=1,J  ★Jの値より処理量が変化
A(I,J)=…
ENDDO
ENDDO
```




```
【IF文を含むループ】不均等例2
DO J=1,M  ←並列化
DO I=1,N
IF (L(I,J)) THEN  ★L(I,J)の値により
…                 処理量が変化
ENDIF
ENDDO
ENDDO
```


FUJITSU

フラットMPIの性能特性
（プロセス数増加時の課題）

フラットMPIの性能特性

FUJITSU

フラットMPIで実行した場合のプロセス数増加における性能面への課題は以下。

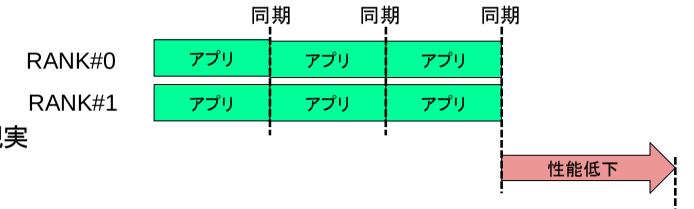
- 通信のオーバヘッド(時間)増加
- アプリケーションが利用できるメモリ量減少
- OSジッタの影響

＋　最内ループに対しては、やはりSIMD化のプログラミングが必要。

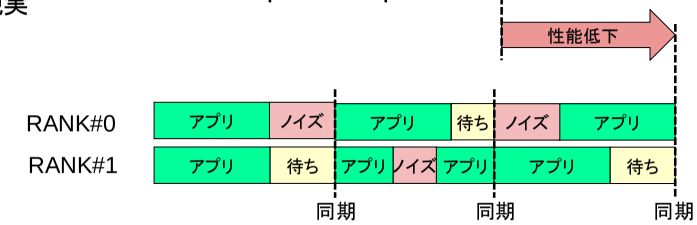
OSジッターについて

FUJITSU

- ノード毎にバラバラに発生するOSノイズ(システム処理)が、ノード間の同期処理によって、アプリ全体として累積し、実行性能を低下させる現象
  - 理想



  - 現実



➔ハイブリッド並列化によってプロセス並列数を減少させることで、OSジッターの影響を低減させることが可能

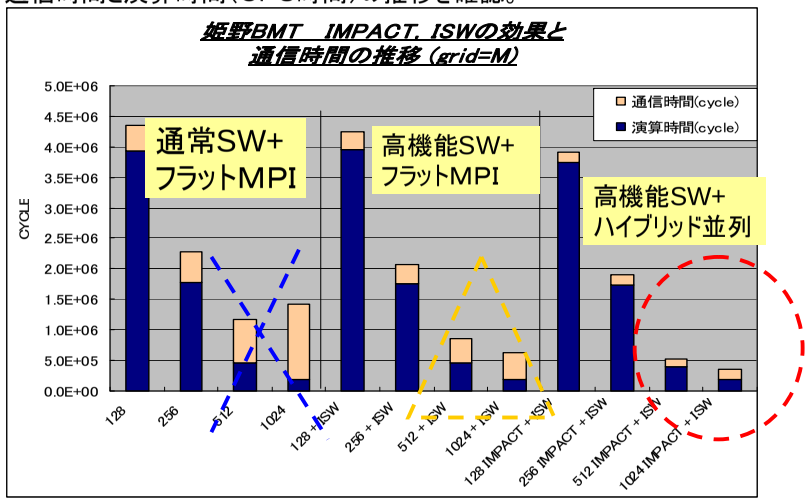
FUJITSU

効果例

姫野BMT：FX1での性能検証

FUJITSU

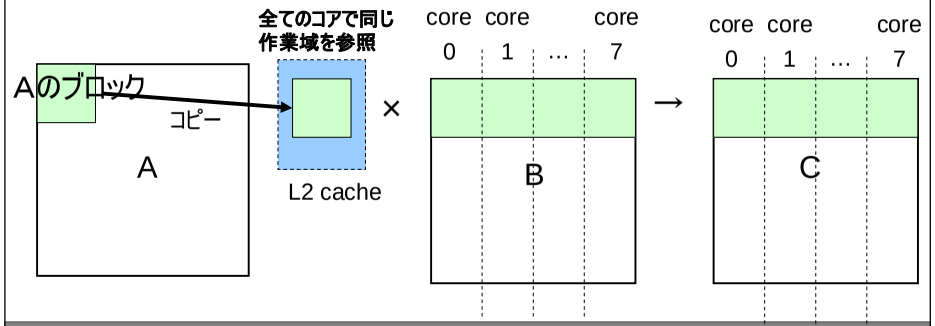
- MPI多次元分割、通信はMPI_allreduce使用、最内ループを自動並列。
  通信時間と演算時間(CPU時間)の推移を確認。



LINPACKのDGEMM：共有キャッシュを利用

FUJITSU

- 行列積AB→Cの計算では、行列をブロックに分けて計算
  - 行列Aのブロックを L2 cacheに乗せる。ブロックが大きいほうが高効率
  - →フラットMPIではL2 cacheサイズがブロックが実質 1/コア数 となる→効率Down
  - →スレッド並列では、ブロックを共有してL2 cache 全体を活用→効率Up
- 効果
  - DGEMMで～2%の性能向上を推定



FUJITSU

まとめ

プログラミングモデル まとめ

FUJITSU

- **富士通は、ハイブリッド並列プログラミングを推奨。**
  →ハードウェア/ソフトウェアの両面で利用者を支援
  狙い：MPIでのみ記述されたフラットMPI実行アプリを自動でハイブリッド並列化へ（自動化できない部分のみOpenMPディレクティブで記述）
- **フラットMPIについても否定するものではない。**
  ただし、通信オーバーヘッドに注意する必要あり。

FUJITSU

【補足資料】
高機能スイッチの位置づけと効果

2009.5.23（2010.1.12改版）
富士通株式会社　清水 俊幸

高機能スイッチの位置づけと効果

- 高並列アプリケーションの実行性能を高めるために，高機能スイッチを開発した
  - 高機能スイッチの概要については，2008年2月18日にPSIシンポジュームのプロトタイプの発表を参照

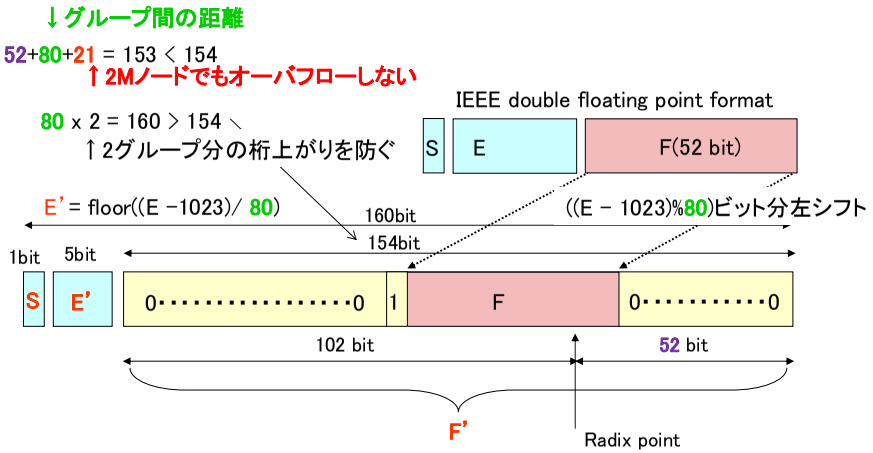
    http://www.psi-project.jp/images/event/toshiyuki_shimizu_20080218.pdf
- 高機能スイッチの位置づけと効果について考察した
  - 実装方式と将来性について分析し，ハード量は少なく、将来の超並列システムへの埋め込みが容易であることを確認した
  - 演算誤差について高機能スイッチが採用した1パス演算方式は最も誤差が小さいことを確認した
  - 実行速度のバラツキについて，ソフトによる1対1通信による実現の測定結果より，メッセージハンドリングを多数伴うソフトインプリは実行速度バラツキが大きいことを確認した
- 今後の取り組み
  - 大規模アプリケーションでの効果の確認などを行う

コレクティブ通信高速化方法比較

	Host上のソフト	NIC上のマイクロ	高機能スイッチ
中継回数	O(log(N))	O(log(N))	O(1)
OS処理による通信処理中断の影響	あり	なし	なし
課題	レイテンシ削減 上記影響軽減	レイテンシ削減	ハードコスト低減 マルチユーザ対応
備考	ハード追加なし	ハード追加なし Myrinetなどでの実装例	専用ハード BlueGeneで実現（2パス）

- PSI 2-1では、新アルゴリズム考案により、原理的に従来方式の2倍高速化を実現
- 高機能スイッチ実現に必要なハード量は少なく、将来の超並列システムへの埋め込みが容易

2Mノードまでサポートするデータフォーマット



- S: 符号
- E': 変換後の指数部（グループを表現）
- F': 変換後の仮数部（固定小数点として計算可能）

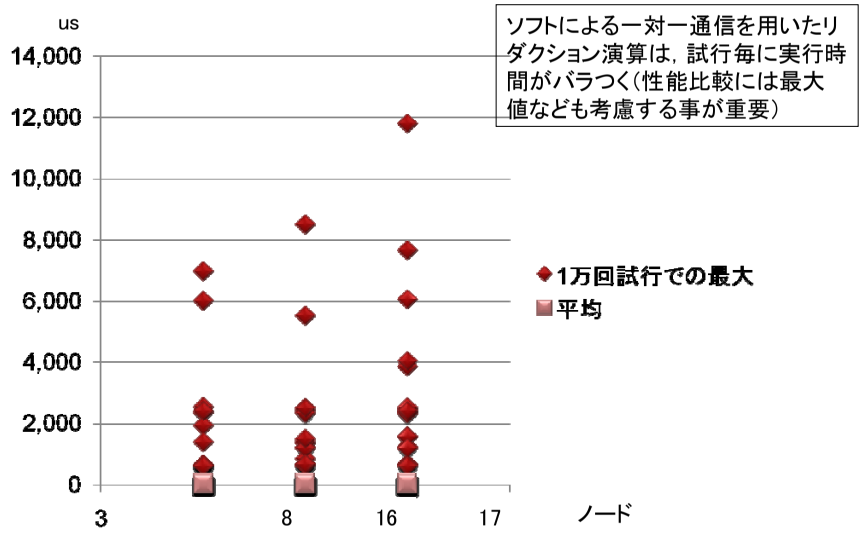
演算方式による誤差（有効ビット数）

データ数	2K	16K	128K	1M	式
IEEE倍精度	50	50	49	49	54-log(logN)
2パス方式	43	40	37	34	54-log(N-1)
1パス方式	54	54	54	54	Min(81-log(N-1),54)

- 有効ビット数は、1パス方式＞IEEE倍精度＞2パス方式となる
- 2パス方式では、2k/16k/128k/1Mデータについてそれぞれ64/67/70/73ビットの計算を実施すると想定
- 1パス方式では、154ビットの計算を実施すると想定（2Mノードまで適用可能）

ソフトリダクションのバラツキ

FUJITSU



ソフトによる一対一通信を用いたリダクション演算は，試行毎に実行時間がバラつく（性能比較には最大値なども考慮する事が重要）

参考：オーバーヘッドのオーダー(BG/L)

FUJITSU

事象	時間のオーダー	備考
キャッシュミス	100ns	
TLB ミス	100ns	
IO 割り込み	1us	ネットワーク含む
PTE ミス	1us	“minor fault”
タイマー割り込み	1us	
ページフォールト	10us	“major fault”
スワップイン	10ms	DISK IO有り
プリエンプション	10ms	プロセス切り替え

■出典：“Exploring Petascale Linux Operating Systems – ZeptoOS”
http://www.cs.unm.edu/~fastos/06meeting/ZeptoOS-PImtg-BostonUsenix-2006.pdf
■ZeptoOS は、Argonne National Lab. が開発中の BG/L ハード上で動作する軽量Linux

4. ベンチマークジョブサブ WG

4.1. サブ WG まとめ

ベンチマークジョブサブ WG まとめ役
上智大学　　南部　伸孝

ベンチマークジョブサブ WG では、利用者に役立つ情報を提供できるベンチマーク(以降、SS 研 BMT と呼ぶ)を目指し、ベンチマークの骨子を決定する活動を行ってきた。一般的なベンチマークでは、演算性能(浮動小数点演算，flops)を評価対象としているが、利用者にとって有意な情報か疑問があった。この点について討論を重ね、利用者に有益な情報として「実行時間」が重要であるとの結論に至った。その主な理由は，一般利用者の計算とは，浮動小数点演算のみならず、入出力性能などの計算に関連するすべての計算要素が複雑に関連するからである。つまり、全てが評価されなければ、演算のみが高速に終了しても意味がない場合が多々見受けられる。さらに、このような複雑な要素が、複数重なって実際には利用者のバッチ利用によって処理されており、単純な浮動小数点演算性能では意味をなさないのが現実である。

SS 研 BMT の特徴は、実際のシミュレーションプログラムを用いて、実行時間を測定することである。利用者は、短時間で演算結果を得ることを求めており、演算性能という間接的な観点ではなく、実行時間という直接的な観点を評価対象としている。また、主要な演算部分にはタイマーを挿入することを想定しており、プログラムのコスト分布なども得られ、挙動を把握することができる。さらに、プログラムの実行時間を予測するという観点も導入する。これは、小規模問題を実際に実行し、大規模問題および多重複数処理における実行時間を予測するというものである。このような観点は既存のベンチマークには無く、SS 研 BMT 特有な観点であり、利用者に有益な情報であると確信している。

また、消費電力という観点も導入し、運用管理者の視点も導入する。これにより、直接プログラムを扱う研究者だけでなく、より多くの人に開かれたベンチマークを目指している。

4.2. 実行性能と消費電力のベンチマーク

上智大学　南部 伸孝
富士通株式会社　軽部 行洋、石附 茂

SS 研 BMT の目的、特徴、活用方法、展開方法、BMT 項目、作成指針について述べる。

1. 目的

利用者の視点に立ち、プログラム性能を評価するためのベンチマークとする。さらに、消費電力に関するベンチマークも行えるものとする。

2. 特徴

演算の様子を再現した擬似コードではなく、実際に意味のあるシミュレーションを行い、プログラム全体を評価対象とする。また、各分野のプログラムを用意し、多くの研究者が利用できる。さらに、消費電力も計測し、運用管理者にも利用価値のあるベンチマークとする。

3. 活用方法

以下に示す 5 項目の観点で活用できるようにする。

1) マシンの基本性能の把握
2) 大規模問題のシミュレーション時間の予測
3) マシン毎のプログラム性能の比較
4) 最適な入出力パターンの検討
5) アプリケーションと消費電力の関係

4. 展開方法

SS 研ホームページで公開する。追加プログラムを募集し、随時更新する。

5. ベンチマーク項目

以下の 4 項目について評価できるものとする。

1) 基本性能（CPU，メモリ，ネットワークに関する性能）
2) プログラム性能（実際のシミュレーションプログラムの性能）
3) I/O 性能（ファイル入出力の性能）
4) 消費電力

6. ベンチマークプログラムの作成指針

ベンチマークの作成にあたり、以下の指針を設定する。

・基本性能および I/O 性能について
  1) 公開されている BMT を利用する
・プログラム性能について
  1) メモリサイズ（問題規模）を簡単に変更できること
  2) 入力データは機種依存性の無いテキスト形式とする
  3) コンパイラ依存性が無いこと
  4) 性能指標の考え方に一貫性があること

7. 公開ベンチマークの利用

基本性能および I/O 性能の測定には、以下の**表** 1 で示す一般に公開されているベンチマークを使用する。

表 1. 公開ベンチマーク

項目	アプリ名	公開 URL
メモリバンド幅	STREAM	http://www.cs.virginia.edu/stream
ネットワーク性能	IMB	http://software.intel.com/en-us/articles/intel-mpi-benchmarks
I/O 性能	IOR	http://ior-sio.sourceforge.net
並列特性	NAS-parallel	http://www.nas.nasa.gov/Resources/Software/npb.html

8. 性能指標の考え方

システムエンジニアの観点ではなく、ユーザの観点で評価する。ユーザにとって役立つ情報として、実行時間を念頭に置き、以下の 4 項目を把握できるベンチマークを作成する。

1) どの処理に時間を要しているか
2) 実行時間に占める、演算・通信・入出力の割合
3) 大規模問題の実行時間予測
4) ハイブリッド並列化の有効性

9. 測定項目と評価値一覧

マシンの基礎性能測定から消費電力までを網羅する総合的なベンチマークを準備し、利用者が希望する項目を選択する方式とする。ベンチマークの選択項目を**表 2**2 に示す。また、ベンチマーク項目の内、プログラム性能に関しては、さらに詳細な測定項目があり、プログラムの挙動を把握できるようにする。プログラム性能に関するプログラム挙動把握のための測定項目を表 3 に示す。

表 2. ベンチマーク選択項目

項目	内容	評価値
基礎性能	各種の公開ベンチマークを利用する ・メモリバンド幅（STREAM） ・通信バンド幅（IMB） ・通信レイテンシ（IMB） ・MPI と OpenMP の比較 （NAS-parallel-benchmark を使用し、MPI 版と OpenMP 版を比較する）	バンド幅（byte/s） 遅延時間（秒） 倍率
IO 性能	入出力性能を計測する ・入出力バンド幅	IO バンド幅（byte/s）
プログラム性能	ユーザプログラムの総合性能を評価する ・演算性能（ハイブリッドを含む） ・通信性能（ハイブリッドを含む） ・IO 性能	実行時間（秒） 演算性能（FLOPS） バンド幅（byte/s）
消費電力	消費電力を評価する ・プログラム実行時の消費電力を検出	演算当りの消費電力(w/flop) 時間当りの消費電力(w/hour)

表 3. プログラム挙動把握のための測定項目

項目	内容
コスト分布 (主要処理の実行時間)	実行時間の長い処理をルーチン単位で表示する。 どのルーチンに時間を要しているかを把握できる。
演算・転送・入出力の割合	全実行時間に占める、演算の割合と転送の割合と入出力の割合を表示する。
大規模問題を想定した場合の実行時間予測	小規模問題を実際に実行して、大規模問題の実行時間を予測する。
ハイブリッドの有効性	flat-MPI とスレッド+MPI で実行時間を比較する。

10. 各測定項目の課題と対策案

ベンチマークを作成するにあたり、解決しなければならない課題がある。**表** 4 に、課題と対策案をまとめる。今後、具体的な対処法を決定する必要がある。

表 4. ベンチマーク測定項目の課題と対策案

項目	課題	対策案
基礎性能 IO 性能	一般に公開されているベンチマークを使用するため、以下 3 点の課題がある。 1) インストール方法 2) 実行方法 3) 出力結果の見方	インストールと実行に関しては、実際のコマンド例を示した解説書を作成する。 出力結果の見方として、各出力項目の意味を解説した資料を作成する。
プログラム性能	1) 主要処理の計算内容を解説し、入力データを用意する必要がある。 2) コンパイラ依存性を除去するため、言語仕様に準拠するよう修正が必要。 3) 各ルーチンに実行時間測定用の時間計測ルーチンを挿入する必要がある。 4) ハイブリッド化を実施する必要がある。	1)と 2)に関しては、プログラム提供者に対応して頂く。 3)と 4)に関しては、サブ WG のメンバーで対応する。
消費電力	消費電力の計測方法を調査する必要がある。また、測定対象として、CPU だけで良いか、ディスクやファンの電力も含めるか等、さらに検討が必要。	有識者に問い合わせ、計測対象と計測方法を詰める。

4.3. ベンチマーク実行例

ベンチマーク構成のイメージを**図** 1 に示す。利用者は、入力パラメータを設定し、制御プログラムを起動する形式を想定している。制御プログラムは、入力データで指定したベンチマークを自動的に起動するように制御するためのプログラムである。

次節で、プログラム性能に関する、ベンチメークの実行イメージを示す。

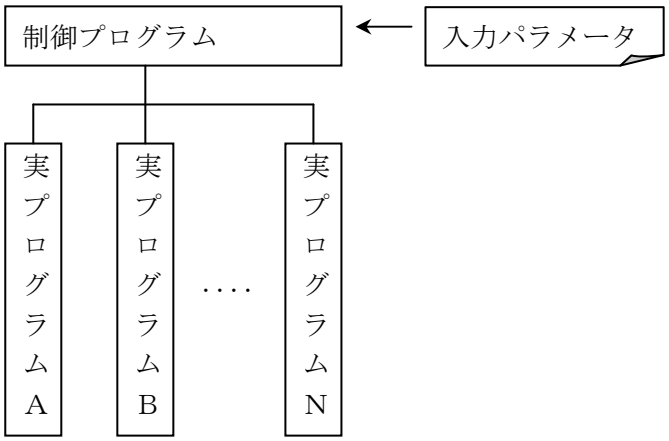


図 1. ベンチマーク構成イメージ

1. 実行例

入力データで、ab_initio_mo を選択した場合の BMT 出力例を**図** 2 に示す。入力パラメータと主要処理の実行時間が出力される。

```
 ----- ab_initio_mo method -----
 -----      [ c6h6 ]       -----
 --------------------------------
 ---- parameter information ----
  number of atom = 12
  number of electron = 42
  number of base function = 36
  threshold in SCF = 1.000000000000000e-05

 ---- execution information ----
  SCF iteration = 67

------ Time information ------
          normalize: 0.970e-04 (sec)
       one_elec_int: 0.388e-01 (sec)
       two_elec_int: 0.111e+02 (sec)
                SCF: 0.210e+01 (sec)
           file I/O: 0.271e+01 (sec)
 Total execute time: 0.159e+02 (sec)
 --------------------------------
```

図 2. ab_initio_mo 選択時の実行結果例

2. 活用例

1) ベンチマークの実行結果からプログラムの特性を把握する。

図 2 から、シミュレーション対象物質がベンゼン(c6h6)であり、基底関数は 36 という情報が得られる。この場合の主要計算は 2 電子積分であり約 11 秒、ファイル出力も約 3 秒要していることが分かる。

さらに、表 5 に示すような、プログラムの特性を解説書に記述し、入力パラメータと演算数の関係を明らかにすることで、実行時間を予測することも可能となる。

表 1. ab_initio_mo の処理概要

プログラム名	処理内容	評価値	備考
ab_initio_mo	[分子軌道法プログラム] 1 電子積分，2 電子積分，SCF 計算を行う。 2 電子積分で扱う積分タイプは、7 種類である。 (ssss , psss , ppss , psps , pspp , ppps , pppp) 1 電子積分の演算量は、$O(N^2)$となり、 2 電子積分の演算量は、$O(N^4)$となる。 SCF 計算は、$O(N^3)$である。 （N=基底関数の数）	実行時間	言語：Fortran77 行数：3565 step 並列化：無(逐次)

上記の解説を元に、以下の情報を得ることができる。

2) 大規模問題の実行時間予測
主要計算である 2 電子積分は、基底関数の 4 乗オーダの演算となる。よって、
[実行時間 ≒ 11(秒)×(想定する基底関数の個数/36)^4]
と予測することができる。

上記のように、プログラムの特性を把握でき、かつ、異なる問題に対する予測もできる点が、一般的なベンチマークとは異なり、ユーザの視点に立ったものとなっている。

以上

５．おわりに

本 WG の最終年度（2009 年）は次世代スーパーコンピュータ計画にとって激動の年であった。5 月に日本電気がベクトル計算機部の開発からの撤退を表明したことは広く HPC 関係者にとって衝撃を与えた。ベクトル計算機実現の夢が潰えたことによる茫然自失(？)、官が進めるプロジェクトから民が撤退するという前代未聞(？)の事態、昔の栄華(？)を知る者にとっては感無量、などさまざまな想いが交錯する状況であった。救いは、富士通がプロジェクトの継続を力強く進めていること、敢えて言えば複数社による複数システムの開発というプロジェクトとしては必ずしも好ましくない桎梏から解放されたことか。

これだけに止まっていれば単に HPC コミュニティにおける「コップの中の嵐」程度の話題に過ぎなかったが、夏の衆議院選挙において下馬評通り政権交代が起き、これまでの官の事業を全面的に見直すとして実施された「事業仕分け」の俎上に次世代スーパーコンピュータプロジェクトが乗せられ、「来年度の予算計上の見送りに限りなく近い縮減」という評価を下されたことが大激震を起こした。“公開”事業仕分けのあり方についてはさまざまな意見があるものの、テレビで実況中継されたこの報道によりスーパーコンピュータや HPC という言葉が人口に膾炙するようになり、一般国民にまで知名度を広げたことは瓢箪から駒の宣伝効果であった。またプロジェクトの凍結を避けるために HPC 関係者が総動員されてその必要性、有用性を一般国民も含めて訴えかける行動にでたことは目を瞠るものがあった。SS 研にあっても 2009 年度の合同分科会終了後に緊急アピールの記者会見を開いた。この年の合同分科会開催地が、次世代スーパーコンピュータが整備される神戸市だったこともあり地元での関心も高かった。

禍を転じて福と為す。これまでは、ややもすれば研究者の唯我独尊の世界に陥りがちであった HPC 社会も改めてその足下を見直し、何をどのように通じて世の中の役に立つのか、個々に問い直し実践する契機となれば幸いである。不安は「喉元過ぎれば熱さ忘れる」や「羹に懲りて膾を吹く」といった類の結果に陥らないか、である。志を大きく持つことがそのような弊害から逃れる道と心したいものである。

さて本題の当 WG 活動 3 年間を振り返ってであるが、当初目論んだ個別テーマの選定は活動期間中にでも随時にサブ WG 化して進めること、は必ずしも十全に実現することができなかった。報告書には 3 つのサブＷＧ報告が掲載されているが、やはり性能評価に重点がおかれ、性能評価という切り口を通じた「プログラミングモデルサブ WG」であり「ベンチマークジョブサブ WG」活動であった。これは一つには多くのサブ WG が立ち上がると事務局や報告者としての富士通の負担が大きくなることにも起因していると思われる。そのような意味から事務局機能の一部をユーザーが担ったり、報告者としてもっと積極的にユーザーが関わる、ということも考えなければいけないであろう。一方、報告書の内容については参加者各位の協力のおかげで SS 研会員各位の役に立つものであると自負している。会員各位の今後の活動に活用していただければ幸いである。

最後になるが、貴重な時間を割いて3年間のWG活動に参加して頂いた会員の皆さん、富士通の担当者、WGの円滑な運営に尽力して頂いたSS研事務局の皆さんに感謝する。

（HPC 技術 WG まとめ役　福田正大）

SS 研 HPC 技術 WG (2007/5-2010/5) 成果報告書

【発行】 サイエンティフィック・システム研究会

【編集】 HPC 技術 WG

本資料に関するお問合せは、下記連絡先へお願いします。
＜連絡先＞サイエンティフィック・システム研究会（SS 研） 事務局
〒105-7123 東京都港区東新橋 1-5-2
TEL：03-6252-2582 FAX：03-6252-2798
Email：office@ssken.gr.jp
http://www.ssken.gr.jp/MAINSITE/

2010 年 5 月 21 日発行