科	学	技	術	計	算	分	科	会		選	出

科学技術計算分科会 2012 年度会合 より

HPC 分野における GPU 活用技術

－アクセラレータ技術 WG 成果報告－

井上 弘士

(九州大学)

サイエンティフィック・システム研究会　　2012 年 10 月 24 日
科学技術計算分科会 2012 年度会合

HPC 分野における GPU 活用技術
～アクセラレータ技術 WG 成果報告～

SS 研 アクセラレータ技術 WG
とりまとめ役：井上弘士（九州大学）

[アブストラクト]
本講演では、2009 年からおよそ 2 年に渡り実施したアクセラレータ技術 WG の成果を報告する。特に、アクセラレーション技術として注目を集めている GPU の活用に焦点をあて、本 WG における主な成果の 1 つである「今後の GPU 活用のあるべき姿」を探るために実施した評価実験について報告する。

[キーワード]
アクセラレーション、GPU、ハイパフォーマンスコンピューティング、コンパイラ、チューニング

１．はじめに

スパコンシステムの計算性能は、これまで概ね 3 年で 10 倍のペースで向上してきた。近年はプロセッサの動作周波数を向上させることが技術的に難しくなったため、プロセッサコアの単体性能を向上させるのではなく、スパコンを構成するノード数を増やす、プロセッサのコア数を増やす（マルチコア化）など、数を増やすアプローチで性能向上のペースを維持してきた。その中で、近年注目されているのがアクセラレータである。汎用プロセッサは、HPC 以外の様々な用途での使用を想定しており、様々なプログラムで簡単に安定した性能が得られる様に実装面積の半分以上をキャッシュメモリとして使用しており、演算ユニットの占める面積比率は低い。それに対して、アクセラレータは HPC 用途にフォーカスしたプロセッサで、キャッシュメモリなどの非演算ユニットのリソース量を最小限に抑え、多数の演算ユニットを搭載して、演算ユニットの面積比率を高めることで、演算性能を高めるアプローチのプロセッサである。アクセラレータは高い演算性能を持つ反面、プログラム開発が難しいなど、その性能を引き出すのは簡単でない。実際に、特定アプリでは高い性能が得られるが、それ以外のアプリでは高い性能が得られないという問題がある。しかし、エクサスケール世代では飛躍的に電力あたり演算性能を高める必要があるため、汎用プロセッサと比べて電力あたり演算性能の優れたアクセラレータ技術は、エクサスケール世代の必須技術であると期待されている。

このような背景の下、SS 研アクセラレータ技術 WG を立ち上げ、アクセラレータとして近年普及しつつある GPU を事例として、将来のアクセラレータのあるべき姿を模索・議論した。特に、GPU に代表されるアクセラレータ上でのプログラム開発の現状とその課題、また、どうすればアクセラレータで高い性能を得られるのか、具体的な事例を用いて調査した結果を報告する。

２．WG 活動概要

WG の活動を支えたメンバーを以下に示す。活動期間中に 8 回の会議を実施した。これらの会議においては、各メンバーによるアクセラレーション技術に関する技術報告に加え、各種実験トライアルの結果報告と議論、さらには、今後のアクセラレーション技術（大規模センター等での運用技術も含む）に関する議論を展開した。

	氏名	所属
担当幹事	村上 和彰	九州大学
推進委員	青木 尊之	東京工業大学
	遠藤 敏夫	東京工業大学
	黒川 原佳	理化学研究所
	滝沢 寛之	東北大学
	伊野 文彦	大阪大学
	井上 弘士	九州大学
	本田 宏明	九州大学
	堀田 耕一郎	富士通(株) 次世代テクニカルコンピューティング開発本部
	丸山 拓巳	富士通(株) エンタプライズサーバ事業本部
	坂口 吉生	富士通(株) TC ソリューション事業本部
	佐々木 啓	富士通(株) TC ソリューション事業本部
	久門 耕一	(株)富士通研究所
	成瀬 彰	(株)富士通研究所

３．おわりに

本 WG の活動において、様々な知見を得ることができた。特に、1)市販コンパイラを用いた「手軽な」GPU の活用においては、ディレクティブの挿入に 2～3 時間程度費やすだけで、人手でのチューニングと比較して 40%程度の性能を得た。また、2)一般には「GPU に不向き」とされるアプリケーションにおいても、実装アルゴリズムを GPU 向けに再検討することで大幅な性能向上を達成できた。今後は、アプリ開発者と実装チューニングの連携、GPU－CPU の協調実行、などが重要になる事が分かった。

HPC分野におけるGPU活用技術
～アクセラレータ技術WG成果報告～

SS研 アクセラレータ技術WG
取りまとめ役 井上弘士（九州大学）

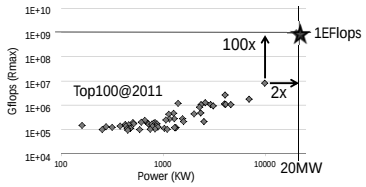
発表内容

- ワーキンググループ（WG）設置の背景
- WG活動概要
- GPUを「手軽に」使って性能が改善するか?
- GPUに不向きと言われるアプリは本当に不向きなのか?
- 将来のアクセラレータ活用のあるべき姿は?

WG設置の背景

エクサ実現へのチャレンジ

- エクサ・フロップス級スパコンの実現における最大の課題は?
  →「消費電力の壁」
- 1EFlops@20MWスパコンを実現するには?
  →「京」を基準にすると消費電力2xで性能100x



“DARPA IPTO Exascale Computing Study”

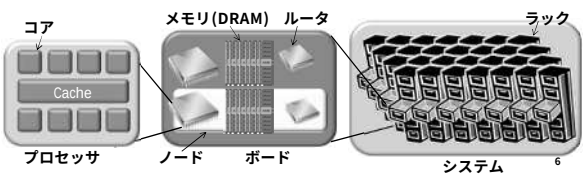
http://www.darpa.mil/tcto/docs/ExaScale_Study_Initial.pdf

- 既存マシンをベースにしたスケーリングによりシステム性能を予測
- 消費電力と面積（最大ラック数）制約を考慮

ノード名	説明	システム例	プロセッサ例
Heavy Node (HN)	高性能プロセッサ搭載型ノード	Red Storm	Intel, AMDなどのプロセッサ
Light Node (LN)	組込み向けプロセッサ搭載型ノード	Blue Gene/L, /P	PPC440, 450
Aggressive Node (AN)	1EFlopsのシステムを前提とした仮想ノード	無し	無し

“DARPA IPTO Exascale Computing Study”

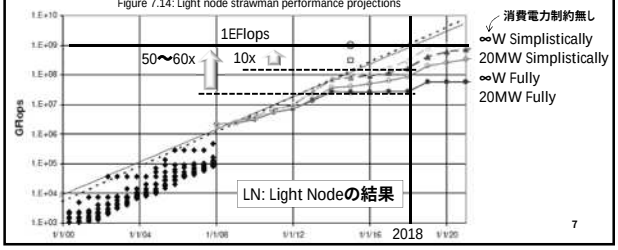
- 半導体デバイス
  - ITRS@2006
- メモリ総容量
  - ∝理論ピーク性能
- ノード消費電力モデル
  - Simplisticlly Scaled Power Model
    - プロセッサ→ITRS@2006
    - 主記憶（DRAM）→∝＃DRAMチップ
    - ルータ→ベースと同じ 割合で一定
  - Fully Scaled Power Model
    - プロセッサ→ITRS@2006
    - 主記憶（DRAM）→∝＃DRAMチップ×性能
    - ルータ→∝性能



“DARPA IPTO Exascale Computing Study”

- 単純な量的変化だけではエクサ@2018は到達できない!
  - Fullyで50～60x，Simplisticallyで10xの性能向上が必要
- 何らかの質的な変化が必要では?
  - アクセラレーション技術

Figure 7.14: Light node strawman performance projections



本WGの狙い

- 現在最も有望なアクセラレーション技術→GPU
  - 「GPU向きアプリ」を対象とした多くの事例報告
- 素朴な，でも，本質的な疑問
  - GPUを「手軽に」使って性能が改善するか?
    - PGIコンパイラを用いた「ライトな」チューニング
  - GPUに不向きと言われるアプリは本当に不向きなのか?
    - GPUに不向きとして知られるアプリを対象に「ヘビーな」チューニング
- 上記の調査を通して将来のアクセラレータ活用のあるべき姿を模索・議論

WG活動概要

メンバーと活動内容

	氏名	所属
担当幹事	村上 和彰	九州大学
推進委員	青木 尊之	東京工業大学
	遠藤 敏夫	東京工業大学
	黒川 原佳	理化学研究所
	滝沢 寛之	東北大学
	伊野 文彦	大阪大学
	井上 弘士	九州大学
	本田 宏明	九州大学
	堀田 耕一郎	富士通(株)
	丸山 拓巳	富士通(株)
	坂口 吉生	富士通(株)
	佐々木 啓	富士通(株)
	久門 耕一	(株)富士通研究所
	成瀬 彰	(株)富士通研究所

会合	日時	活動内容
第一回	2009年 10月29日(木)	• WG全体スケジュールの議論 • 今後の議題検討
第二回	2010年 1月19日(火)	• 会員報告 (PGIアクセラレータコンパイラ、コーンビーム再構成の高速化、初心者によるGPUプログラミング事例、複合型計算機向けソフトウェア開発環境) • 富士通報告 (GPGPUプログラミングの状況、GPUSIMから見たGPU)
第三回	2010年 4月7日(水)	• 会員報告 (Linpack on GPU搭載スパコンTsubame) • 富士通報告 (OpenCLの評価) • OpenCLに関する議論
第四回	2010年 7月14日(水)	• 会員報告 (PGIコンパイラ評価) • 富士通報告 (nVidia新GPU評価)
第五回	2010年 12月3日(金)	• 会員報告 (アクセラレータの大規模システム導入課題、Linpack on Tsubame2)
第六回	2011年 3月25日(金)	• 会員報告 (PGIコンパイラ評価) • 富士通報告 (分子軌道法プログラムのGPU化検討)
第七回	2011年 8月19日(金)	• 会員報告 (CUDA Fortran評価) • 富士通報告 (分子軌道法プログラムのGPU化結果)
第八回	2012年 1月19日(木)	• まとめ

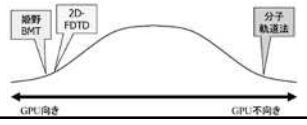
GPUを「手軽に」使って性能が改善するか?

3人の開発者による PGIアクセラレータ利用事例

- PGIアクセラレータ：
  - nVidia GPU向けのディレクティブ挿入方式のプログラム開発言語・環境
- 3人の開発者

開発者A	OpenMP・MPIによる並列プログラム開発経験はあるが，PGIアクセラレータの使用経験は無し．
開発者B	計算科学分野のアプリ開発経験は豊富．逐次プログラム開発が主体であり，プログラムの並列化はあまり詳しくない．
開発者C	並列処理のエキスパートであり，CUDA・OpenCLでのプログラム開発経験がある．

- 3種類のアプリ
  - 姫野BMT
  - 2D-FDTD
  - 分子軌道法



実験結果

	姫野BMT			2D-FDTD			分子軌道法
	GFLOPS	Directive数	開発工数	GFLOPS	Directive数	開発工数	GFLOPS
PGI ver.1	14.9	2行	1時間 (学習に数日)	18.5	10行		GPUコード 生成できず
PGI ver.2	19.2	8行	+30分	…	…	…	…
CUDA版 (by 開発者C)	50	…	数日	…	…	…	…
CUDA版 (by 開発者B)	…	…	…	45.5	…	3日程度	…
Xeon X5670 2.93GHz 1コア	3.9	…	…	7.0	…	…	…
Xeon X5670 2.93GHz 4コア	7.8	…	…	…	…	…	…

エキスパート（開発者C）のCUDA版実装と比較して40%程度の性能

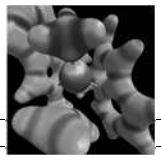
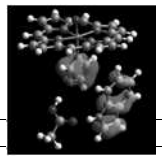
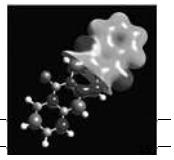
13

GPUに不向きと言われるアプリは本当に不向きなのか?

14

分子軌道法とは

- 分子内において，主に電子がどのような運動をしており，どのようなエネルギーを持っているかを量子化学的計算により求める方法の一つ
  - 分子物性の解析
  - 創薬、新素材の開発
  - ex) プリンタのカラーインク、液晶ディスプレイ

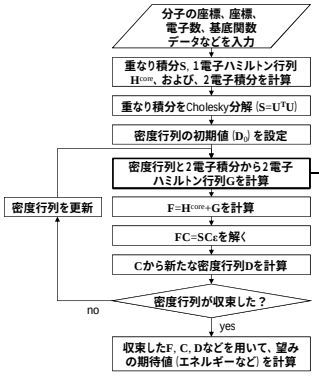
GPUでの高速化は難しい「らしい」

分子軌道法プログラムの GPU 化は一般的に難しいとされている
- 数例の報告のみ

- オリジナルコード
  - コードが複雑、ステップ数大
  - PGI アクセラレータコンパイラのディレクティブの方法では、GPU へのオフロードコードは生成されず
- GPU使用の問題点
  - 倍精度計算の使用
  - 開平逆数、指数関数計算が必要
  - 条件分岐計算
  - レジスタ量が不十分
  - シェアードメモリを利用のための適切な方法が不明確
  - デバイスメモリサイズの制限
  - などなど

16

分子軌道法プログラムの処理フロー



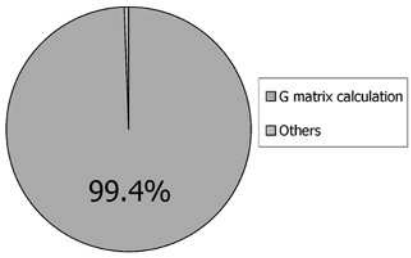
- 積分駆動型アルゴリズムによる２電子フォック行列計算
- Obara のアルゴリズムによる積分計算法計算
- 全てC言語で記述

$$G_{ij} = \sum_{k=1}^{N}\sum_{l=1}^{N} D_{kl}\left[2\left\langle \varphi_i\varphi_j \middle| \varphi_k\varphi_l \right\rangle - \left\langle \varphi_i\varphi_k \middle| \varphi_j\varphi_l \right\rangle\right]$$

2電子積分計算 (ERI) を含むG行列計算式

17

ボトルネックはG行列計算



- CPU
  - Core-i7 2600K 3.4GHz (1core)
- Compiler
  - Intel v12.1 (-O3)
- Program
  - Kyudai-HFR
- Input data
  - h16o8 (24 atoms)
- Elapsed time
  - 14.29 sec

分子軌道法プログラムの高速化
G行列計算の高速化

18

CPU向け並列化

G行列計算の概要＆ OpenMPによるCPU向け並列化

係数行列作成
1st loop begin
2nd loop begin
3rd loop begin
4th loop begin
初期積分計算
垂直漸化式計算
4th loop end
3rd loop end
水平漸化式計算
G行列更新
2nd loop end
1st loop end

omp parallel for

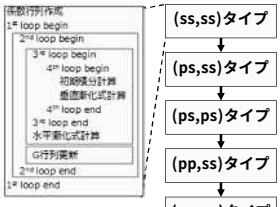
1st loopのiterationをスレッドに分配

omp atomic

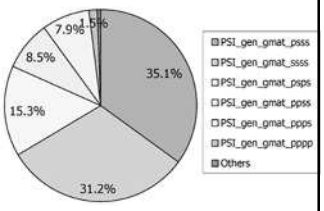
他スレッドと同じ箇所を更新する可能性

```
G [ ij ] += 4 * x * D [ kl ]
G [ kl ] += 4 * x * D [ ij ]
G [ ik ] -=     x * D [ jl ]
G [ il ] -=     x * D [ jk ]
G [ jk ] -=     x * D [ il ]
G [ jl ] -=     x * D [ ik ]
```


実際には6種類の積分計算



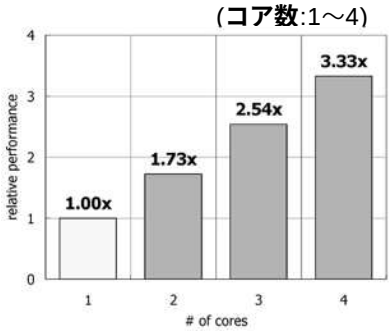
(ss,ss)タイプ
(ps,ss)タイプ
(ps,ps)タイプ
(pp,ss)タイプ
(pp,ps)タイプ
(pp,pp)タイプ



- CPU: Core-i7 2600K 3.4GHz (1core)
- Compiler: Intel v12.1 (-O3)
- Program: Kyudai-HFR
- Input data: $(H_2O)_8$ (24 atoms)
- Elapsed time: 14.29 sec

OpenMPによる並列化効果

(コア数:1～4)



- CPU
  - Core-i7 2600K 3.4GHz (1-4 cores)
- Compiler
  - Intel v12.1 (-O3 -openmp)
- Program
  - Kyudai-HFR
  - 6種類の積分型を全てOpenMPで並列化
- Input data
  - h16o8 (24 atoms)

性能

積分型	実行時間（1コアに対する性能向上）		
	CPU		GPU
	1コア	4コア	
(ss,ss)	246 (1.0x)	66.2 (3.7x)	?
(ps,ss)	279 (1.0x)	85.8 (3.3x)	
(ps,ps)	122 (1.0x)	39.6 (3.1x)	
(pp,ss)	67 (1.0x)	21.3 (3.1x)	
(pp,ps)	62 (1.0x)	21.0 (3.0x)	
(pp,pp)	12 (1.0x)	4.0 (3.0x)	

GPU向け並列化（NAÏVE実装）

G行列計算処理 ((ss,ss)積分型)

係数行列作成
1st loop begin
  2nd loop begin
    3rd loop begin
      4th loop begin
        初期積分計算
        垂直漸化式計算
      4th loop end
    3rd loop end
    水平漸化式計算
    G行列更新
  2nd loop end
1st loop end

係数行列作成
係数行列数：16種
総量：*2.4MB* (入力依存)

1st & 2nd loop：
ループ回数：*139,656* (入力依存)

ERI計算：
ループ回数：*81* (入力依存)
ERI値：*1種* (積分型依存)

G配列更新
前出ERI値と関連する部分を更新
G行列更新：*6箇所* (積分型依存)
G配列要素数：*1,596* (入力依存)

(*) Input data: $(H_2O)_8$ →24 atoms

G行列計算のGPGPU化

係数行列作成
1st loop begin
  2nd loop begin
    3rd loop begin
      4th loop begin
        初期積分計算
        垂直漸化式計算
      4th loop end
    3rd loop end
    水平漸化式計算
    G行列更新
  2nd loop end
1st loop end

1st & 2nd loop：
ループ回数：*139,656* (入力依存)

◆1st & 2nd loopの各iterationを1スレッドに割当

(*) Input data: h16o8 (24 atoms)

G行列計算のGPGPU化

係数行列作成
1st loop begin
  2nd loop begin
    3rd loop begin
      4th loop begin
        初期積分計算
        垂直漸化式計算
      4th loop end
    3rd loop end
    水平漸化式計算
    G行列更新
  2nd loop end
1st loop end

ERI計算：
ループ回数：*81* (入力依存)
ERI値：1種 (積分型依存)

◆マルチスレッド化
◆各ERI値の計算には5スレッド使用（定量的な解析より）

(*) Input data: h16o8 (24 atoms)

ボトルネックはどこに?

係数行列作成
1st loop begin
  2nd loop begin
    3rd loop begin
      4th loop begin
        初期積分計算
        垂直漸化式計算
      4th loop end
    3rd loop end
    水平漸化式計算
    G行列更新
  2nd loop end
1st loop end

Nスレッド並列
(*) N=2～81

N-1スレッドは待機
（6回のatomic操作）

(*) Input data: h16o8 (24 atoms)

G行列計算のGPGPU化

係数行列作成
1st loop begin
  2nd loop begin
    3rd loop begin
      4th loop begin
        初期積分計算
        垂直漸化式計算
      4th loop end
    3rd loop end
    水平漸化式計算
    G行列更新
  2nd loop end
1st loop end

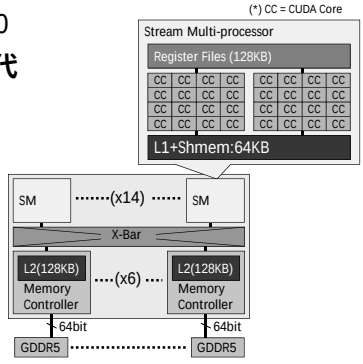
部分G行列更新
ERI値毎に、関連する箇所を更新
G行列更新箇所：*6*
(全要素数：*1,596*)

◆更新箇所は不連続
◆他スレッドが同時に同じG行列要素を更新する可能性
◆atomicCAS()による排他処理
  ◆CAS: Compare And Swap

(*) Input data: h16o8 (24 atoms)

GPU環境

- Tesla C2050
  - Fermi世代
- CUDA 4.2



性能

積分型	実行時間（1コアに対する性能向上）			
	CPU		GPU	
	1コア	4コア	Naïve	
(ss,ss)	246 (1.0x)	66.2 (3.7x)	45 (5.4x)	?
(ps,ss)	279 (1.0x)	85.8 (3.2x)	---	
(ps,ps)	122 (1.0x)	39.6 (3.0x)	---	
(pp,ss)	67 (1.0x)	21.3 (3.1x)	---	
(pp,ps)	62 (1.0x)	21.0 (2.9x)	---	
(pp,pp)	12 (1.0x)	4.0 (3.0x)	---	

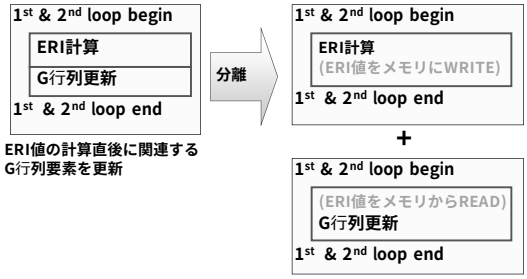
GPU向け並列化
（実装アルゴリズムの最適化）

ボトルネックはどこに存在する?
～atomic操作を外すと…～

- CPU (1core) 246 ms
- GPU (2x2x5) 45 ms (5.4x)
- GPU (w/o atomic操作) 25 ms (9.8x)
  (*)ただし、正しい結果は得られない

ボトルネックはG行列計算のatomic操作
→これをなんとかしないと速くならない!

Kernel分離：atomic操作を無くす！



ERI値の計算直後に関連する
G行列要素を更新

ERI値をメモリに記録するオーバーヘッドは
高いメモリバンド幅で隠蔽(と期待)

性能

積分型	実行時間（1コアに対する性能向上）				
	CPU		GPU		
	1コア	4コア	Naïve	カーネル分割	
(ss,ss)	246 (1.0x)	66.2 (3.7x)	45 (5.4x)	10.5 (23.4x)	?
(ps,ss)	279 (1.0x)	85.8 (3.2x)	---	16.2 (17.2x)	
(ps,ps)	122 (1.0x)	39.6 (3.0x)	---	12.6 (9.7x)	
(pp,ss)	67 (1.0x)	21.3 (3.1x)	---	6.3 (10.6x)	
(pp,ps)	62 (1.0x)	21.0 (2.9x)	---	10.1 (6.1x)	
(pp,pp)	12 (1.0x)	4.0 (3.0x)	---	9.6 (1.3x)	

性能

積分型	実行時間（1コアに対する性能向上）				
	CPU		GPU		
	1コア	4コア	Naïve	カーネル分割	
(ss,ss)	246 (1.0x)	66.2 (3.7x)	45 (5.4x)	10.5 (23.4x)	GPU向き •繰り返し多 •計算単純
(ps,ss)	279 (1.0x)	85.8 (3.2x)	---	16.2 (17.2x)	
(ps,ps)	122 (1.0x)	39.6 (3.0x)	---	12.6 (9.7x)	
(pp,ss)	67 (1.0x)	21.3 (3.1x)	---	6.3 (10.6x)	
(pp,ps)	62 (1.0x)	21.0 (2.9x)	---	10.1 (6.1x)	
(pp,pp)	12 (1.0x)	4.0 (3.0x)	---	9.6 (1.3x)	GPU不向き •繰り返し少 •計算複雑

・積分型により性能UP率に大きな違い
・複雑な積分型はGPUで性能向上難しい?
→さらなるチューニングを試みる!

積分型によるG行列計算処理の違い

係数行列作成
1st loop begin
2nd loop begin
3rd loop begin
4th loop begin
初期積分計算
垂直漸化式計算
4th loop end
3rd loop end
水平漸化式計算
G行列更新
2nd loop end
1st loop end

計算されるERI値
(ss,ss) 1個
(pp,pp) 81個

更新されるG配列要素
(ss,ss) 最多 6箇所
(pp,pp) 最多486箇所

ERI計算の複雑さは積分型次第

積分型	# used registers	Shmem (bytes)	Spill stores (bytes)	Spill loads (bytes)
(ss,ss)	54	0	0	0
(ps,ss)	63	0	64	64
(ps,ps)	63	0	296	220
(pp,ss)	63	0	164	136
(pp,ps)	63	0	852	636
(pp,pp)	63	0	2,708	2,332

- レジスタ数が足りない→レジスタスピル発生

Shmemの活用

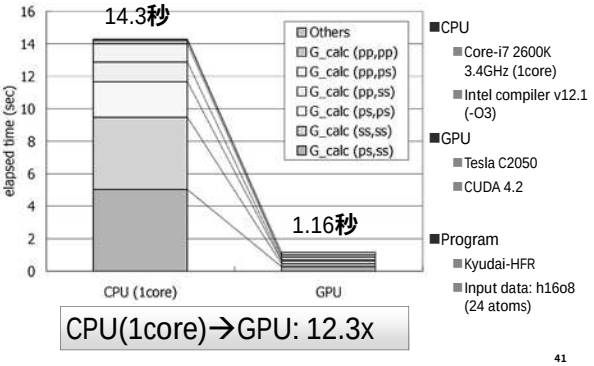
積分型	# used registers	Shmem (bytes)	Spill stores (bytes)	Spill loads (bytes)
(ss,ss)	50	0	0	0
(ps,ss)	63	0	28	28
(ps,ps)	62	4,224	164	116
(pp,ss)	63	768	224	44
(pp,ps)	63	7,680	700	448
(pp,pp)	63	7,872	232	44

- Shmemを使ってレジスタスピル量を削減
  - 他にも細かなチューニング

性能

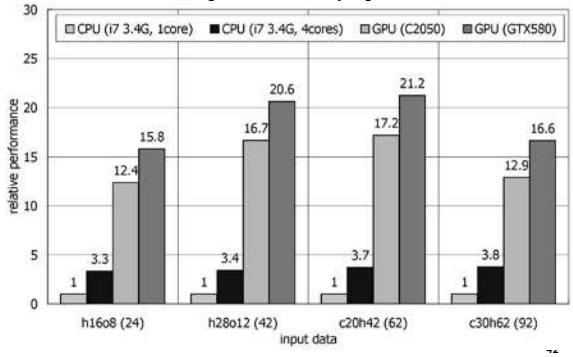
積分型	実行時間（1コアに対する性能向上）				
	CPU		GPU		
	1コア	4コア	Naïve	カーネル分割	更なるチューニング
(ss,ss)	246 (1.0x)	66.2 (3.7x)	45 (5.4x)	10.5 (23.4x)	10.5 (23.4x)
(ps,ss)	279 (1.0x)	85.8 (3.2x)	---	16.2 (17.2x)	14.9 (18.8x)
(ps,ps)	122 (1.0x)	39.6 (3.0x)	---	12.6 (9.7x)	8.6 (14.1x)
(pp,ss)	67 (1.0x)	21.3 (3.1x)	---	6.3 (10.6x)	4.2 (15.9x)
(pp,ps)	62 (1.0x)	21.0 (2.9x)	---	10.1 (6.1x)	9.2 (6.8x)
(pp,pp)	12 (1.0x)	4.0 (3.0x)	---	9.6 (1.3x)	7.6 (1.5x)

全体性能比較(CPU vs GPU)

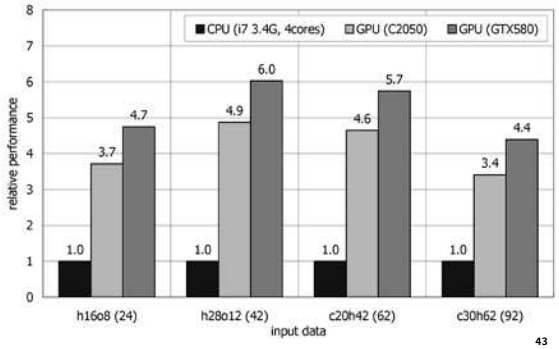


CPU(1core)→GPU: 12.3x

各種入力データでの性能比較（vs. 1コア）



各種入力データでの性能比較（vs. 4コア）



将来のアクセラレータ活用のあるべき姿は?

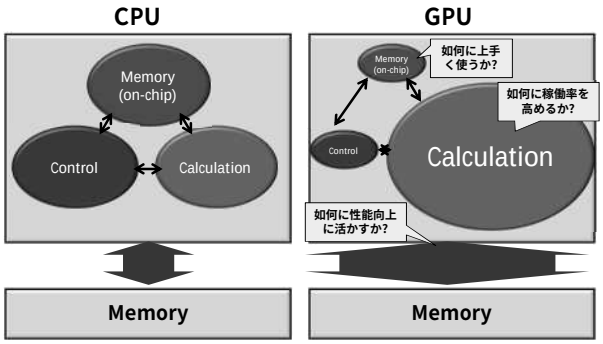
GPUを「手軽に」使って性能が改善するか?

- GPUに向いている単純なコード
  - 数行のディレクティブ挿入
  - エキスパートによるチューニング実装と比較して40%程度の性能
- コンパイルできないコードも存在

GPUに不向きと言われるアプリは本当に不向きなのか?

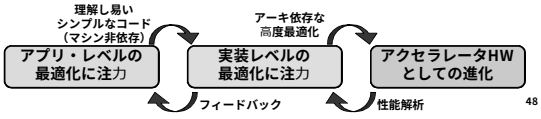
- 実装アルゴリズムをGPU特性に合わせて見直すことでCPUを大きく上回る性能を達成
  - GTX 580性能：対CPU(1core)で最大21.2倍
    対CPU(4core)で最大6.0倍
- 本トライアルで性能向上を達成できた理由
  - 理解し易いシンプルなコードの提供
  - 実装屋の存在

CPU vs. GPU



アクセラレータ「活用の」あるべき姿（1/2）

- GPU化に「向く」部分と「向かない」部分はある
  - 依然としてアプリ屋と実装屋のギャップは大きい
- アクセラレータ活用のあるべき姿
  - 素性の良いアプリ→コンパイラが「そこそこ」使える
  - 向かないアプリ→アプリ屋と実装屋の協調（Co-Design?）で多くを「向く」アプリに!



アクセラレータ「活用の」あるべき姿（2/2）

- CPU(Fat Core)とGPU(Thin Cores)の併用
  - どうしてもGPUに向かない処理はある（pppp型など）
  - ただし，データサイズが小さく実行時間が短いことが多い
  - このような処理はCPUに任せる
- タスク並列的なCPU/GPU併用

科 学 技 術 計 算 分 科 会 選 出
科学技術計算分科会 2012 年度会合 より

スーパーコンピュータ「京」での MPI の実装と評価

三浦 健一
(富士通株式会社)

サイエンティフィック・システム研究会
科学技術計算分科会 2012 年度会合
2012 年 10 月 24 日

スーパーコンピュータ「京」での MPI の実装と評価

三浦 健一
富士通株式会社 次世代テクニカルコンピューティング開発本部 ソフトウェア開発統括部

[アブストラクト]
スーパーコンピュータ「京」[1]の共用が 9/28 から開始された。「京」で使用されている MPI ライブラリでは 8 万ノード規模の並列計算に耐えられ、かつ高速通信ができるよう、様々な工夫が施されている。本発表においては、「京」における MPI ライブラリの特徴である、省メモリ通信、通信最適化、最適ランク配置、低レベル通信ライブラリの実装を説明し評価する。また併せて、アプリケーションへの適用事例に関して報告する。

[キーワード]
Tofu インターコネクト，MPI 通信，大規模並列，省メモリ，最適ランク配置

1 はじめに

スーパーコンピュータ「京」の共用が 9/28 から開始された。「京」は 8 万ノード超のノード群が Tofu(Torus fusion)と呼ばれる 6 次元メッシュ/トーラス型インターコネクトで結合された構成となっている。Tofu では 6 次元メッシュ/トーラスを生かした故障ノード回避や迂回通信、ジョブ単位に論理トーラス割当を行うことが可能となり高い運用性を実現している。また Tofu では 4 個の通信インターフェイスを用いた同時通信や高機能バリアを用いたハードウェア同期を行うことでより高速に通信を行うことが可能である。我々は「京」上で Tofu を用いて通信を行う MPI ライブラリの開発を行い、大規模環境での評価を行った。

2 MPI ライブラリの実装概要

「京」の MPI ライブラリは Open MPI をベースに拡張を行ったが、8 万ノード規模の Tofu インターコネクトへの適用にあたり、以下の点に注意して設計を行った。

- 8 万ノード規模の並列化に耐えうる実装
- Tofu の特性を生かした高速通信

[1] 「京」は理化学研究所の登録商標です。

- ユーザの使いやすさ

MPI 通信を行うためには通信相手毎に送受信バッファが必要となるが、8 万ノード分の送受信バッファを確保していたのでは MPI ライブラリ自体のメモリ使用量が膨大になってしまう。「京」の MPI ライブラリでは省メモリ通信モードと高速通信モードを使い分けることで、省メモリを維持しつつ高速に通信を行うことを狙っている。

Tofu の特性を生かした高速通信では、特に大規模並列計算で問題となる集団通信に関して、Tofu 向けアルゴリズムを新規開発することで、ハードウェア性能を最大限引出す事を狙っている。

またユーザの使いやすさとして、通信を行うランクを近くに配置することで性能チューニングを簡単に行うことができる RMATT(Rank Map Automatic Tuning Tool)、突き放し通信や 4 個の通信インターフェイスを使った並列通信を容易に記述することができる「拡張 RDMA インターフェイス」の設計・実装を行った。

3　MPI ライブラリの評価

省メモリ通信では 8 万ノード規模の並列処理においてデフォルトメモリ使用量を 400MB 程度まで抑えることができた。通信最適化においては集団通信の Tofu 専用アルゴリズムの採用により既存アルゴリズムに比べ 5～10 倍程度の性能向上が確認できた。RMATT を用いた最適ランク配置では、NAS parallel Benchmark に適用することで 7%の性能改善が確認できた。「拡張 RDMA インターフェイス」では 4 個の通信インターフェイスを用いた通信を行うことで性能改善の効果が確認できた。

アプリケーションへの適用事例としては昨年度ゴードン・ベル賞を獲得した RSDFT や 4 部門全てで 1 位を獲得した HPCC の Global FFT や Global RandomAccess の通信ライブラリとして使用され高効率の実行に貢献した。

4　まとめ

「京」でサポートしている MPI ライブラリの実装と評価に関して述べた。「京」の運用は始まったばかりであり、今後多くの技術者が「京」に触れて「京」の素晴らしさを体感していただけることを期待している。本 MPI ライブラリを用いることで大規模アプリケーションの開発・運用に少しでも貢献できれば幸いである。

[参考文献]

(1) 住元真司 et al., 「京」のための MPI 通信機構の設計(2012), SACSIS2012

(2) 今出広明 et al., 大規模計算環境のためのランク配置最適化手法 RMATT(2011), SACSIS2011

(3) T. Adachi et al., The design of ultra scalable MPI collective communication on the K computer(2012), ISC'12

FUJITSU
shaping tomorrow with you

スーパーコンピュータ「京」※でのMPIの実装と評価

2012年10月24日
三浦　健一
富士通株式会社

※「京」は理化学研究所の登録商標です。

発表の概要

- 「京」とTofuインタコネクトの概要
  - 理化学研究所と共同で開発した「京」とTofuの特徴を説明
- MPIライブラリの設計方針
- 各項目の課題・実装・評価
  - 省メモリ通信
  - 集団通信高速化
  - 最適ランク配置
- アプリケーション評価
- まとめ

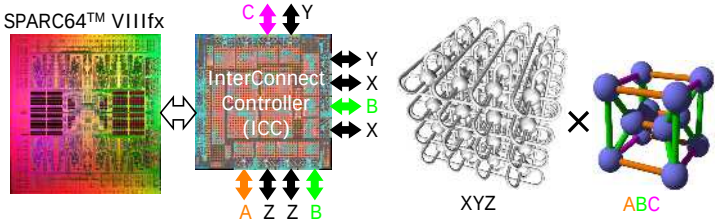
「京」とTofuインタコネクトの概要

スーパーコンピュータ「京」システム概要

Tofuインターコネクト 概要

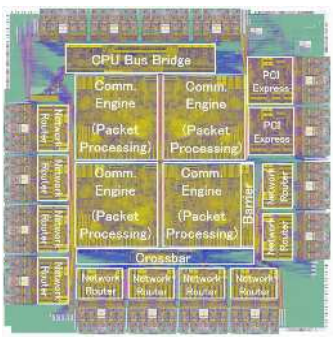
- SPARC64™ VIIIfx専用のノード間インターコネクト
- "**To**rus **fu**sion" 3D-Torus × 3D-Torus = 6D-Torus

ネットワーク・トポロジ	6次元メッシュ／トーラス
座標軸	X, Y, Z, A, B, C
最大ネットワーク・サイズ	32, 32, 32, 2, 3, 2
「京」システム構成	トーラス軸: X, Z, B ／ メッシュ軸: Y, A, C 計算ノード: Z = 1〜16 ／ IOノード: Z = 0



InterConnect Controller（ICC）Chip 概要

設計目標: 高帯域、低遅延、省電力、高信頼性

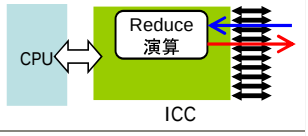


- **基本仕様**
  - RDMAエンジン×4＋高機能バリア
  - ネットワーク次数 10
  - PCI Expressルート機能内蔵
  - クロック 312.5 MHz
- **65nm ASIC**
  - 18.2mm x 18.1mm
  - ゲート数 48M
  - SRAM 12Mbit
  - 差動入出力信号 128レーン
- **高帯域**
  - リンク帯域 5GB/s × 双方向
  - スイッチング容量 140GB/s
- 低遅延
  - Virtual Cut-Through転送 ~100ns

ICCのハードウェア機能

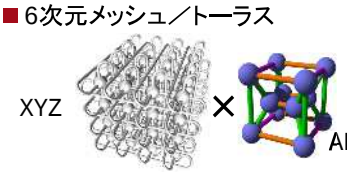
- ポート数10（XYZ軸：6＋ABC：4)、10方向同時ルーティングが可能
- 5GB/sのRDMAエンジンを4個搭載、同時に4送信4受信が可能

- 高機能バリア
  - バリア同期とAllreduce集団通信に対応
    - 64ビット整数: AND, OR, XOR, MAX, SUM
    - 独自160ビット浮動小数点: SUM
  - ソフトウェア実装に比べ最大7.8倍の性能差



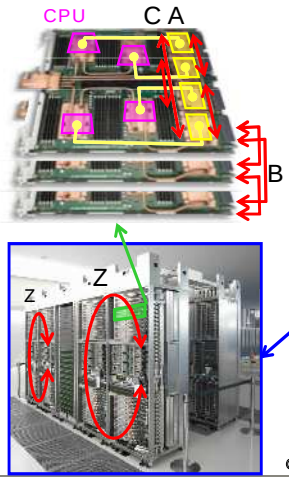
Tofuの概要

- 6次元メッシュ／トーラス

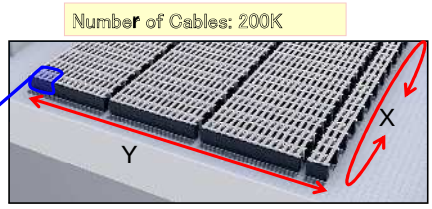


※ インターコネクトLSI仕様はY軸トーラス可能
「京」のシステム実装構造ではY軸メッシュ

Tofuインタコネクトの「京」での実装

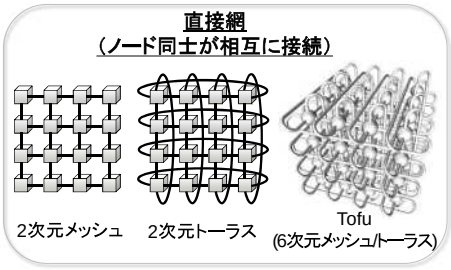


- **Node**: 1 CPU + 1 ICC (Tofu Chip)
- **Topology**: 6D Mesh/Torus
  - Mesh: Y, A, C / Torus: X, Z, B
- A{2} x C{2} on a board
- B{3} x Z{17} in two racks
  - 48 boards: Z=1-16, compute nodes
  - 12 I/O boards: Z=0, I/O nodes



Courtesy of RIKEN

なぜ、6次元トーラスメッシュにしたのか？



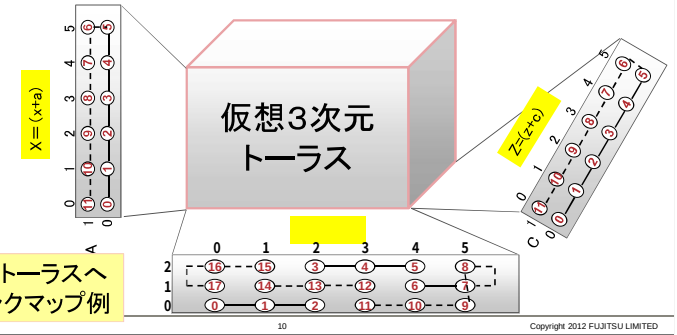
	直接網			間接網
	3次元メッシュ	3次元トーラス	Tofu	Fat-tree
大規模対応	○	○	○	×
信頼性・運用性	×	×	○	○
平均ホップ数	×	△	○**	◎
バイセクション帯域 *	×	△	○(トーラス)	◎

* ネットワークのボトルネック部分の帯域

**Tofuの6次元メッシュ/トーラスはショートカットパスを有する

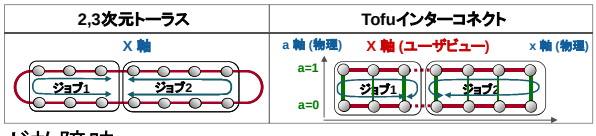
Tofuへの仮想トーラス導入とランクマップ

- TofuのXYZとABCを組み合わせ、仮想トーラス提供
  - 1次元、2次元、3次元トーラスを構成
  - 3次元ランクマップの基本： X=x+a, Y=y+b, Z=z+c
  - 2次元：X=x+a+b, Y=y+z+c、1次元：X=x+y+z+a+b+c

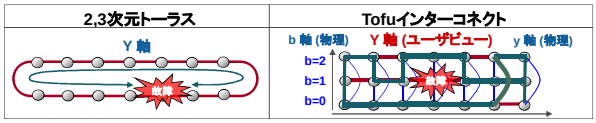


運用性、信頼性比較：2,3次元トーラス vs.Tofu

- 複数ジョブ実行時
  - トーラス: 分割するとメッシュになる
  - Tofu: 分割してもトーラス維持



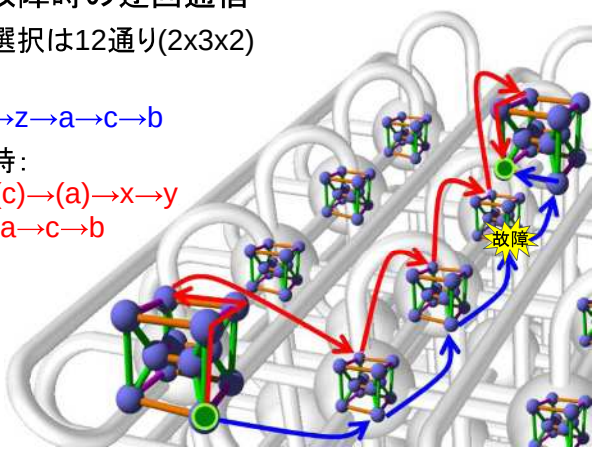
- ノード故障時
  - トーラス: メッシュになる
  - Tofu:故障を回避してトーラス維持



ノード故障時の迂回通信

FUJITSU

- ノード故障時の迂回通信
  - 経路選択は12通り(2x3x2)
  - 通常:
    x→y→z→a→c→b
  - 故障時:
    (b)→(c)→(a)→x→y
    →z→a→c→b



FUJITSU

MPIライブラリの設計方針

MPIライブラリの概要

FUJITSU

- MPI (Message Passing Interface)とは:
  HPCのプログラムで複数の計算機間での様々な通信機能を実現するライブラリの業界標準規格
  京ではMPI-2.1規格採用

- MPI規格の概要
  - 1対1通信
  - 集団通信
  - MPI-IO
  - データタイプ、コミュニケータ、トポロジーなど
- つい最近MPI-3.0規格が2012年9月21日にリリース
  - Non-blocking Collective
  - Neighborhood Collective: BlockingとNon-blocking,
  - One-sided: RDMAを想定したGet,Put,Atomic 等

「京」におけるMPIの設計方針

FUJITSU

- 8万ノード超並列に対応し世界トップ性能を達成するため、1ノード1プロセスでのハイブリッド並列(OpenMP, VISIMPACT)を考える
- MPIの改善項目は多岐にわたるため、ターゲットアプリケーションで使用する通信の性能改善を第1の目標とし、順次改善していく

「京」におけるMPIライブラリの実現目標と要件

FUJITSU

- 目標: 8万ノードクラスの並列処理に耐えうる構造で、実アプリケーションで高い実効性能を達成すること

	アプリケーションと通信面での特徴
SimFold	逐次プログラム
Modylas	Allgatherv, Allreduce
GAMESS	Allreduce, Bcast
RSDFT	Allreduce, Bcast,隣接通信,Reduce,Reduce_Scatter, Allgather
LANS	3次元隣接通信
NICAM	Send-Recv(最大同時通信数19)、通信パターン複雑
LatticeQCD	6方向隣接Send-Recv
HPL	Allgather, Bcast, Send-Recv
NPB-FT	Alltoall

- 要件
  - 上記アプリケーションをペタクラスに引き上げる通信性能実現
  - システムソフトのメモリ利用は総物理メモリの10% (1.6GB)未満を確保

「京」におけるMPIライブラリの実現課題

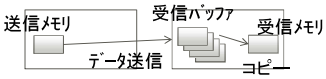
FUJITSU

- 大規模並列時のメモリ問題
  - 既存MPI実装では、8万ノード(8万プロセス)で11.6GBのメモリ使用量、抜本的な対策必須
- 8万ノード規模での高性能通信の実現
  - 集団通信性能: Tofuのハードウェア性能を如何に引き出すか
    - 重要な集団通信: Bcast, Allreduce, Alltoall, Allgather
  - 1対1通信性能: ランクの配置位置が重要
- 本講演では省メモリ通信、集団通信、最適ランク配置を取り上げる

省メモリ通信

MPI通信の実現方法

- Eager通信
  - バッファを用いて通信を行う
  - メリット
    - 同期を取らなくても通信可能
  - デメリット
    - バッファへのコピーによる遅延が発生

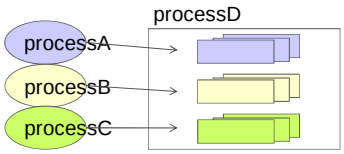


- Rendezvous通信
  - アドレスを通知しRDMA WRITEで直接書き込む
  - メリット
    - ゼロコピー通信が可能。
  - デメリット
    - アドレス通知のための通信が別途必要(Eager通信)

⇒いずれの通信を行うためにも、受信バッファは必要

大規模並列時の課題

- MPI通信のためには受信バッファが必要
  - 通信相手毎に受信バッファを用意することで高速通信を実現
  - 8万ノード規模では11.6GBのメモリを使用。



Open MPIのメモリ使用量
- 1ノード ・・・ 145KB

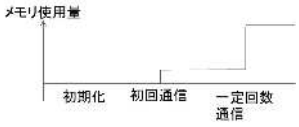
<内訳>
バッファサイズ ・・・145B
バッファ数 ・・・256個
通信インターフェイス数 ・・・4

- 8万ノード ・・・

⇒大規模並列においては受信バッファのメモリ使用量削減が課題

省メモリのための通信方式の実現

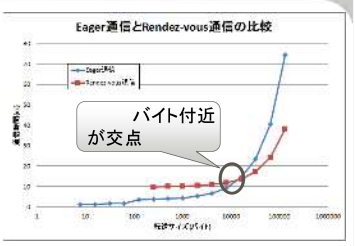
- メモリ使用量と通信性能はトレードオフの関係
  - 一般にメモリ使用量を削減すると通信性能は落ちる
  - ただし、スケールするアプリでは通信相手数が限定
    - ターゲットアプリではほとんどが隣接通信(RSDFT, LANS, LQCD)
    - NICAMでも最大転送相手数は19
- 省メモリ化の実現方針
  - 高性能通信を割り当てるノード数を制限する
- 省メモリ化実現のため： 省メモリ型通信方式を追加
  - 高速型通信(従来型通信) ・・・ 高速通信を行うための十分なメモリ量を確保
  - 省メモリ型通信 ・・・ 通信を行うための必要最小限のメモリ量を確保
- 各通信方式の選択：
  - 初期化時：割り当て無し
  - 通信開始時：省メモリ型通信
  - 頻繁に通信を行うランク間は高速型通信に移行



「京」での省メモリの実装方針

- 省メモリで高速通信実現のために
  - 以下の方針でバッファを割り当てる
    - 高速型通信では理想的な通信とするため1M(16Kx64個)を割り当てる
    - 省メモリ型通信では最低限通信に必要な2K(128※x16個)を割り当てる

※Rendezvousのアドレス交換に必要なバイト数

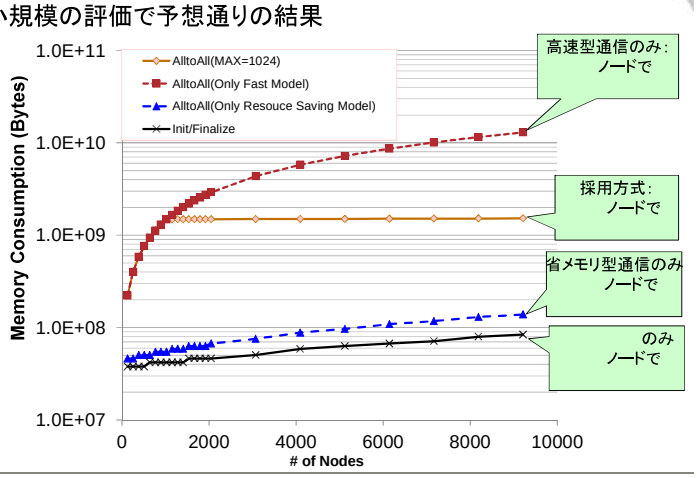


通信方式	バッファサイズ	通信プロトコル		
		~128ﾊﾞｲﾄ	~13.7Kﾊﾞｲﾄ	13.7Kﾊﾞｲﾄ~
高速型通信	1M(16Kﾊﾞｲﾄx64個)	Eager	Eager	Rendezvous
省メモリ型通信	2K(128ﾊﾞｲﾄx16個)	Eager	Rendezvous	Rendezvous

- デフォルトでは15回通信を行った先着1024ランクが高速型通信に移行
  - 初期化時の通信等を考え、ある程度の通信回数で移行を判断
  - 先着1024ランクはメモリ使用量上限(1Gバイト)から逆算
  - バッファサイズ、高速型通信の通信相手数は変更可能

「京」Alltoallを利用したメモリ消費評価

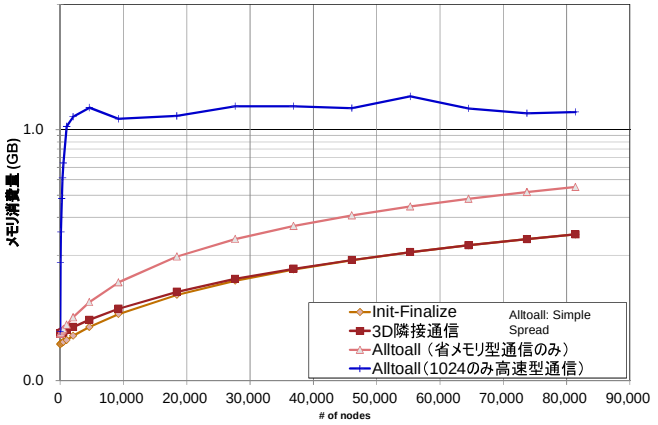
- 小規模の評価で予想通りの結果



「京」の10PF環境での省メモリ性評価

FUJITSU

- 10PF環境においても、目標である総物理メモリの10%を達成



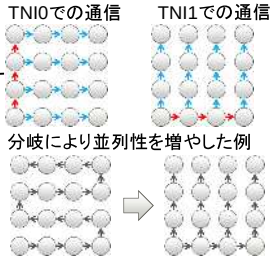
FUJITSU

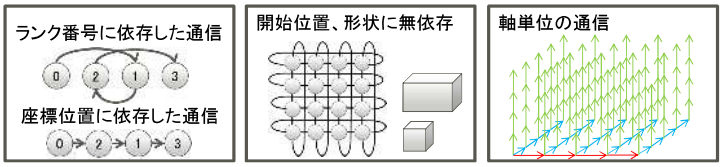
集団通信高速化

Tofuの特性(4TNI,ﾄｰﾗｽ)を活かした集団通信の実現

FUJITSU

- 高速化のための方針
  - 多重度を増やすことによりスループットをあげる
  - 分岐により並列性を増やし中継数を減らす
- アルゴリズム検討時の考慮事項
  - 「京」では仮想次元位置を基準に考える
  - 形状、開始位置に依存しない規則性のあるアルゴリズムとする
  - 基本は軸単位で考える

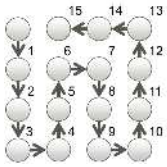




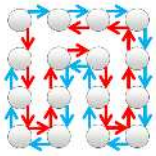
MPI_Bcast(長メッセージ)高速化の考え方

FUJITSU

- 既存アルゴリズムの問題点
  - 1多重のためスループットが出ない
  - 中継数が多いためレイテンシへの影響が大きい
- 高速アルゴリズム実現のためには
  - 高スループットな転送
    - できるだけ多くのTNIを使って多重転送を行う
  - 低遅延な転送
    - 分岐により中継数を減らす
- 課題解決のためには以下が重要
  - 多重パイプライン転送における中継数の削減



既存アルゴリズムの通信手順

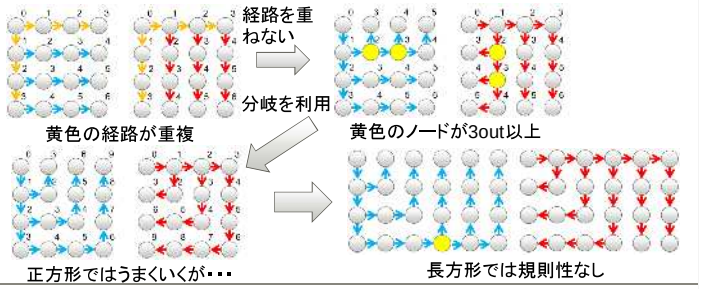


分岐無し2多重ではレイテンシが問題

2次元での2多重パイプライン転送を考える

FUJITSU

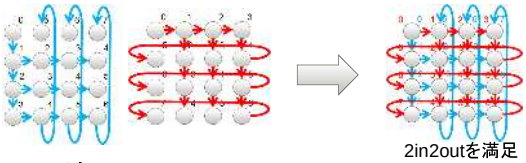
- 要件
  - 4多重転送実現のため2多重パイプライン転送では2in2outを条件とする
- 考え方
  - 拡張性を考え規則性のある配置とする
  - 要件を満足する組合せを探す



3次元アルゴリズム(Trinaryx3)

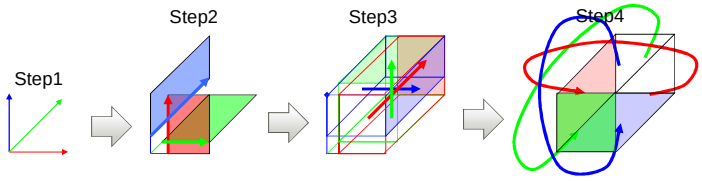
FUJITSU

- 2次元アルゴリズム



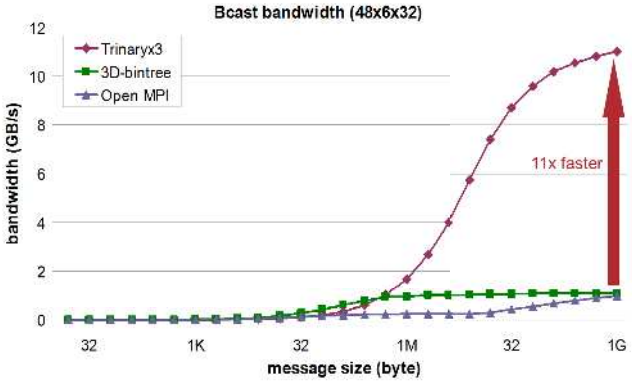
2in2outを満足

- 3次元への適用(Trinaryx3)
  - Tofuの3個のTNIを利用して1/3のデータを3方向に送信
  - トーラス上で効率的に分岐させることでパイプライン段数を削減



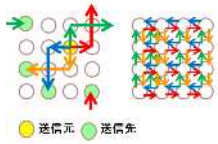
Bcast性能評価結果

- Trinaryx3の適用により既存アルゴリズムに比べて11倍の高速化を達成

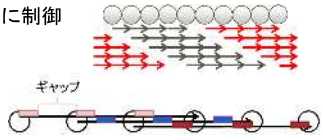


高速集団通信アルゴリズム(MPI_Alltoall)

- MPI_Alltoallの高速化(長メッセージ)
  - 集団通信の中で通信量が最も多く、制御が難しい通信方式
- 既存のアルゴリズムでは、理論性能比で60%程度が限界
  - パケット競合等で通信路を有効に使えていない
  - 通信路をどのように制御するかが課題
- 効率的なリンク制御のためには
  - 全ノードが一斉に同一ホップ転送を行うことで、各リンクにかかる負荷を一定にする

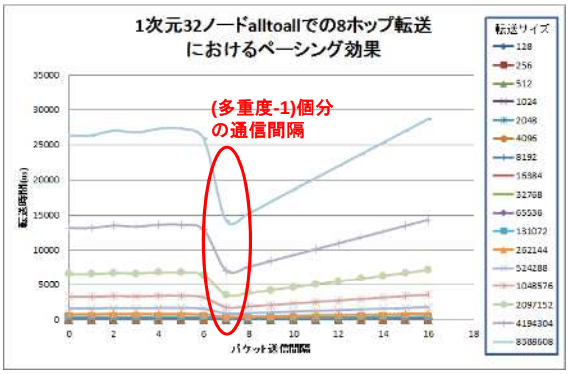


  - パケット衝突によるロスが発生しないように制御
    - パケツリレー方式で各リンクを細かく制御
    - ペーシング技術を活用する(「Tofu」で採用)
      - パケット送信間隔を空けることで衝突を回避

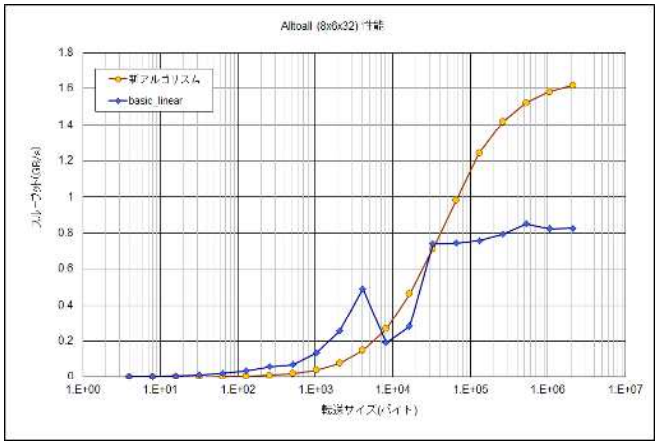


1次元重ね合せ通信のペーシング効果

- 全ランクが一斉に同一ホップ転送を行う場合に、多重度に応じて適切な通信間隔をあけることで理想的に通信できることを確認。

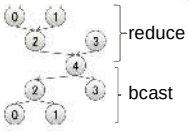


性能結果(8x6x32 alltoall性能)

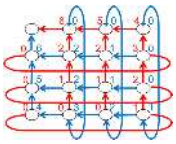


高速集団通信アルゴリズム(MPI_Allreduce)

- MPI_Allreduceの高速化(長メッセージ)
  - Trinaryx3 ReduceとTrinaryx3 Bcastの組合せで実現
  - ReduceはBcastの逆向きの経路で転送を行い、データが揃ったタイミングで演算を行い結果を次のノードに送信

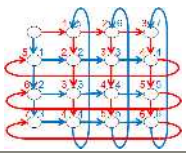


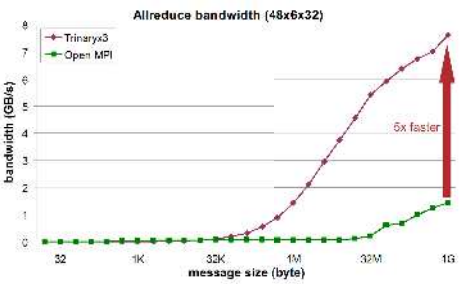
Reduce



\+

Bcast

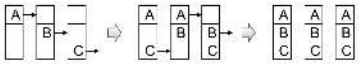


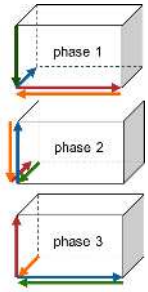


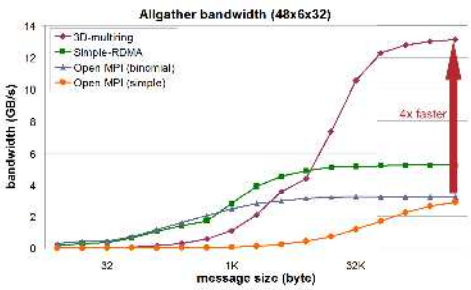
高速集団通信アルゴリズム(MPI_Allgather)

- MPI_Allgatherの高速化(長メッセージ)
  - データを4分割し、4方向に並列送信
  - 各方向毎に1D ringで送信
    phase1：1ノードのデータ
    phase2：軸の全ノードのデータ
    phase3：面の全ノードのデータ

1D ring

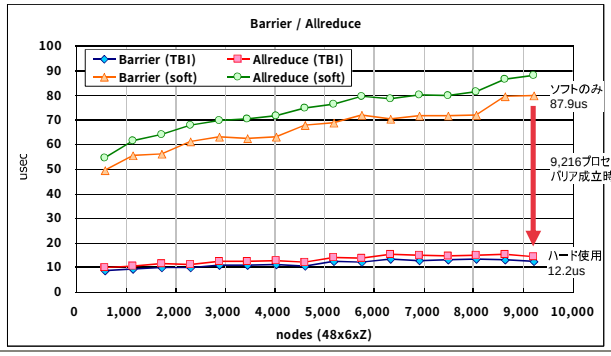






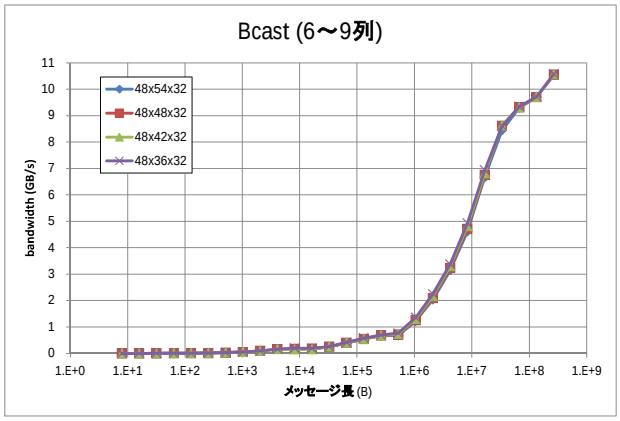
高速集団通信(MPI_Barrier)

- Tofuハードウェアに実装された、高機能バリア機能(TBI)のMPI実装
  - Tofu内仮想ランク+TBI使用により、バリアの高速かつ安定した性能を実現
  - Allreduce(MPI_SUMと固定小数点演算)やBcastにも適用
  - 最大7.2倍の性能差



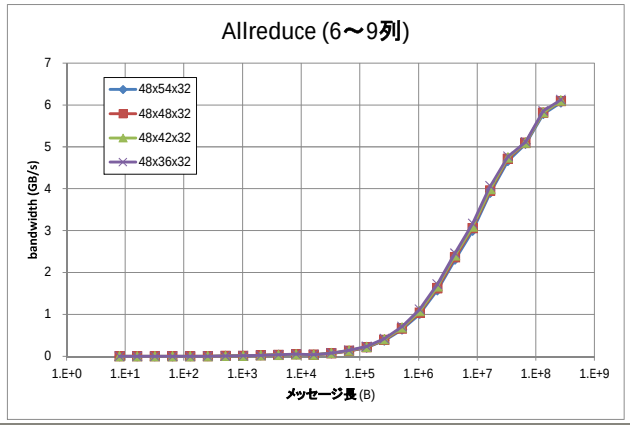
10PF環境での評価(MPI_Bcast)

- 10PF環境で問題なくスケール(中継段数:114段⇒132段)



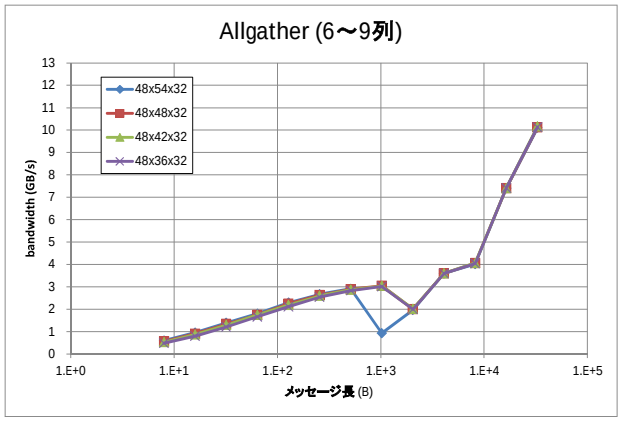
10PF環境での評価(MPI_Allreduce)

- 10PF環境で問題なくスケール(中継段数:114段⇒132段)



10PF環境での評価(MPI_Allgather)

- 10PF環境で問題なくスケール



アプリケーションの効果

アプリケーションへの貢献

- 各種ベンチマークの性能向上に貢献

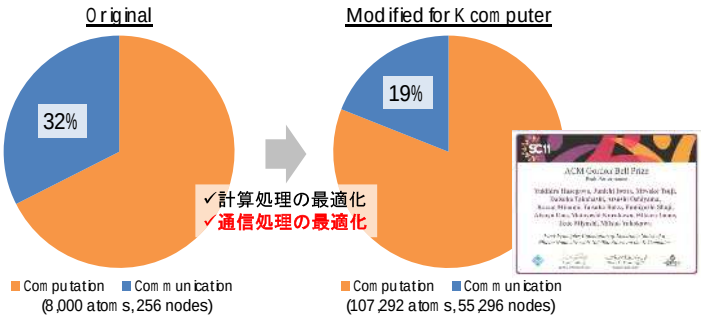
アプリケーション	使用する集団通信	性能
Linpack (線形方程式)	Bcast Allgatherv	実行性能10.51Pflops 実行効率93.17%
RSDFT(実空間第一原理分子動力学計算)	Allreduce	実行性能3.08Pflops 実行効率43.63% (ｺﾞｰﾄﾞﾝﾍﾞﾙ賞獲得)
HPCC(Global FFT) (高速フーリエ変換)	Alltoall	実行性能34.7Tflops* (HPCC全部門制覇)

* : using FFTE5.0β developed by Prof . Takahashi of Tsukuba Univ.

RSDFTにおけるAllreduce最適化の効果

FUJITSU

■ 最適化された***Allreduce*** は2011 Gordon Bell Award*に大きく貢献



◆新しい高速allreduce アルゴリズム：通信スループット 3.2GB/s.
◆アプリケーション性能：3.08Pflops (43.6% efficiency)

* : SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, Article No. 1 „First-principles calculations of electron states of a silicon nanowire with 100,000 atoms on the K computer

Courtesy of RIKEN

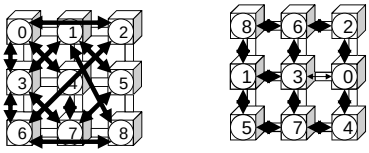
FUJITSU

最適ランク配置

最適ランク配置の狙い

FUJITSU

■ トーラスにおいては通信路が共有されるため、ランクをどのノードに配置するかによりアプリケーション性能が異なる。



■ N並列のランク配置の組み合わせはN!となり、ユーザにとってトーラス上でのランクの配置の最適化は敷居が高い。

■ 頻繁に通信を行うランクを近郊に配置するほうが、通信衝突が発生する可能性が低くなり通信性能がよくなる。

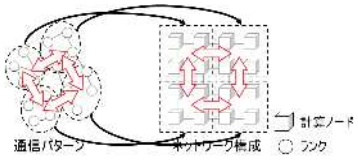
■ ランク間の通信量を元にグループ分けを行い、頻繁に通信を行うランクを近郊に、通信を行わないランクを遠方に配置することで解決を図る

最適ランク配置の手法

FUJITSU

■ 基本的な考え方
  ■ 頻繁に通信を行うランクを近郊に、通信を行わないランクを遠方に配置

■ アルゴリズム
  ■ 2分割の繰り返しによる探索空間の圧縮
    通信量によるグラフ分割を繰り返し行う。この際マッピングされるネットワーク構成を意識して隣接の関係を保ちながら分割を行う。



  ■ 最適化処理

SA: Simulated Annealing ※SAにおける解 ランク配置

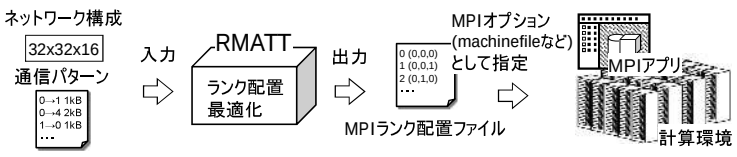
RMATTとは

FUJITSU

■ RMATT(Rank Mapping Automatic Tuning Tool)

ランク間の通信情報を元に、通信量が多いランクをより近郊に配置することで最適なランク配置を決定するためのツール

■ M を用いた通信処理時間のチューニング



チューニングに要する開発者の負担を削減

適用例：NAS Parallel Benchmark

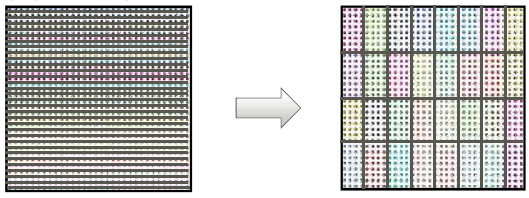
FUJITSU

■ Case 1
  ■ NPROCS=1024, CLASS=B
  ■ 2D Torus 32x32

	最適化なし	M
実行時間	133秒	124秒

実行時間を約　削減

ランク配置最適化の結果



最適化なし(x,y順に配置)　RMATTによるランク配置

本結果は最適解ではないが、どのような方針で配置すれば最適化が可能かの判断にも使用可能

■ Case 2
  ■ NPROCS=8192, CLASS=D
  ■ 2D Torus 128x64

	最適化なし	M
実行時間	10　4秒	秒

通信時間を約64%削減
実行時間を約　削減

まとめ

FUJITSU

- 「京」のMPIは8万ノードクラスの並列処理に耐えアプリケーションの高い実行性能達成を第1目標として開発
  本講演では、特に
  - 省メモリ通信
  - 集団通信高速化
  - 最適ランク配置

  について説明
- 今後は使いやすさや、個々の機能の性能向上を目指し取り組みを行う。

FUJITSU

shaping tomorrow with you

科	学	技	術	計	算	分	科	会		選	出

HPC フォーラム 2012 より

Algorithms and Software in the Post-Petascale Era

William D. Gropp
(University of Illinois)

Algorithms and Software in the Post-Petascale Era

William Gropp
www.cs.illinois.edu/~wgropp

Extrapolation is Risky

- 1989 – T – 23 years
  - Intel introduces 486DX
  - Eugene Brooks writes "Attack of the Killer Micros"
  - 4 years *before* TOP500
  - Top systems at about 2 GF Peak
- 1999 – T – 13 years
  - NVIDIA introduces its GPU (GeForce 256)
    - Programming GPUs still a challenge 13 years later
  - Top system – ASCI Red, 9632 cores, 3.2 TF Peak (about 3 GPUs in 2012)
  - MPI is 7 years old

HPC Today

- High(est)-End systems
  - 1 PF (10^{15} Ops/s) achieved on a few "peak friendly" applications
  - Much worry about scalability, how we're going to get to an ExaFLOPS
  - Systems are all oversubscribed
    - DOE INCITE awarded almost 900M processor hours in 2009; 1600M-1700M hours in 2010-2012; (big jump planned in 2013 – over 5B hours)
    - NSF PRAC awards for Blue Waters similarly competitive
- Widespread use of clusters, many with accelerators; cloud computing services
  - These are transforming the low and midrange
- Laptops (far) more powerful than the supercomputers I used as a graduate student

HPC in 2012

- Sustained PF systems
  - Kei Computer (Fujitsu) at RIKEN, Kobe, Japan (2011)
  - "Sequoia" Blue Gene/Q at LLNL
  - NSF Track 1 "Blue Waters" at Illinois
  - Undoubtedly others (China, …)
- Still programmed with MPI and MPI+other (e.g., MPI+OpenMP or MPI+OpenCL/CUDA or MPI+OpenACC)
  - But in many cases using toolkits, libraries, and other approaches
    - And not so bad – applications will be able to run when the system is turned on
  - Replacing MPI will require some compromise – e.g., domain specific (higher-level but less general)
    - Lots of evidence that fully automatic solutions won't work

End of an Era

- IN THE LONG TERM (~2017 THROUGH 2024) "While power consumption is an urgent challenge, its leakage or static component will become a major industry crisis in the long term, threatening the survival of CMOS technology itself, just as bipolar technology was threatened and eventually disposed of decades ago." [ITRS 2009]
- Unlike the situation at the end of the bipolar era, no technology is waiting in the wings.

The Post-Moore Era

- Scaling is ending
  - Voltage scaling ended in 2004 (leakage current)
  - Feature scaling will end in 202x (not enough atoms)
  - Scaling rate will slow down in the next few years
  - Continued scaling in the next decade will need a sequence of (small) miracles (new materials, new structures, new manufacturing technologies)
- ☛ *Compute Efficiency* becomes a paramount concern
  - More computations per joule
  - More computations per transistor

HPC in 2020-2023

- Exascale systems are likely to have
  - Extreme power constraints, leading to
    - Clock Rates similar to today's systems
    - A wide-diversity of simple computing elements (simple for hardware but complex for software)
    - Memory per core and per FLOP will be much smaller
    - Moving data anywhere will be expensive (time and power)
  - Faults that will need to be detected and managed
    - Some detection may be the job of the programmer, as hardware detection takes power
  - Extreme scalability and performance irregularity
    - Performance will require enormous concurrency
    - Performance is likely to be variable
      - Simple, static decompositions will not scale
  - A need for latency tolerant algorithms and programming
    - Memory, processors will be 100s to 10000s of cycles away. Waiting for operations to complete will cripple performance

Algorithms and Applications Will Change

- Applications need to become more dynamic, more integrated
- System software must work with application:
  - Code complexity (Autotuning)
  - Dynamic resources (no simple PGAS)
  - Latency hiding (Nonblocking algorithms, interfaces (including futures))
  - Resource sharing (more performance information, performance asserts, runtime coordination)

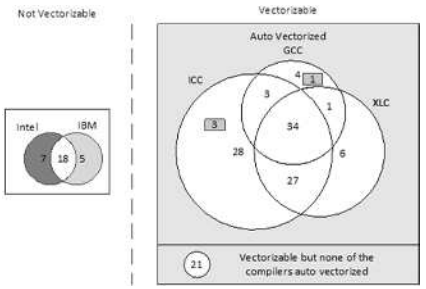
How Do We Make Effective Use of These Systems?

- Better use of our existing systems
  - Blue Waters will provide a sustained PF, but that typically requires ~10PF peak
- Improve node performance
  - Make the compiler better
  - Give better code to the compiler
  - Get realistic with algorithms/data structures
- Improve parallel performance/scalability
- Improve productivity of applications
  - Better tools and interoperable languages, not a (single) new programming language
- Improve algorithms
  - Optimize for the real issues – data movement, power, resilience, …

Make the Compiler Better

- It remains the case that most compilers cannot compete with hand-tuned or autotuned code on simple code
  - Just look at dense matrix-matrix multiplication or matrix transpose
  - Try it yourself!
    - Matrix multiply on my laptop:
    - N=100 (in cache): 1818 MF (1.1ms)
    - N=1000 (not): 335 MF (6s)

How Good are Compilers at Vectorizing Codes?



S. Maleki, Y. Gao, T. Wong, M. Garzarán, and D. Padua. *An Evaluation of Vectorizing Compilers*. PACT 2011.

Media Bench II Applications

Appl	XLC	ICC	GCC	XLC	ICC	GCC
	Automatic			Manual		
JPEG Enc	-	1.33	-	1.39	2.13	1.57
JEPG Dec	-	-	-	-	1.14	1.13
H263 Enc	-	-	-	1.25	2.28	2.06
H263 Dec	-	-	-	1.31	1.45	-
MPEG2 Enc	-	-	-	1.06	1.96	2.43
MPEG2 Dec	-	-	1.15	1.37	1.45	1.55
MPEG4 Enc	-	-	-	1.44	1.81	1.74
MPEG4 Dec	-	-	-	1.12	-	1.18

Table shows whole program speedups measured against unvectorized application

S. Maleki, Y. Gao, T. Wong, M. Garzarán, and D. Padua. *An Evaluation of Vectorizing Compilers*. PACT 2011.

How Do We Change This?

- Test compiler against "equivalent" code (e.g., best hand-tuned or autotuned code that performs the same computation, under some interpretation or "same")
  - In a perfect world, the compiler would provide the same, excellent performance for all equivalent versions
- As part of the Blue Waters project, Padua, Garzaran, Maleki are developing a test suite that evaluates how the compiler does with such equivalent code
  - Working with vendors to improve the compiler
  - Identify necessary transformations
  - Identify opportunities for better interaction with the programmer to facilitate manual intervention.
  - Main focus has been on code generation for vector extensions
  - Result is a compiler whose realized performance is less sensitive to different expression of code and therefore closer to that of the best hand-tuned code.
  - Just by improving automatic vectorization, loop speedups of more than 5 have been observed on the Power 7.
- But this is a long-term project
  - What can we do in the meantime?

Give "Better" Code to the Compiler

- Augmenting current programming models and languages to exploit advanced techniques for performance optimization (i.e., *autotuning*)
- Not a new idea, and some tools already do this.
- But how can these approaches become part of the mainstream development?

How Can Autotuning Tools Fit Into Application Development?

- In the short run, just need effective mechanisms to replace user code with tuned code
  - Manual extraction of code, specification of specific collections of code transformations
- But this produces at least two versions of the code (tuned (for a particular architecture and problem choice) and untuned). And there are other issues.
- What does an application ***want*** (what is the Dream)?

Application Needs Include

- Code must be portable
- Code must be persistent
- Code must permit (and encourage) experimentation
- Code must be maintainable
- Code must be correct
- Code must be faster

Implications of These Requirements

- Portable - augment existing language. Either use pragmas/comments or extremely portable precompiler
  - Best if the tool that performs all of these steps looks like just like the compiler, for integration with build process
- Persistent
  - Keep original and transformed code around: *Golden Copy*
- Maintainable
  - Let user work with original code *and* ensure changes automatically update tuned code
- Correct
  - Do whatever the application developer needs to believe that the tuned code is correct
    - In the end, this will require running some comparison tests
- Faster
  - Must be able to interchange tuning tools - pick the best tool for *each* part of the code
  - No captive interfaces
  - Extensibility - a clean way to add new tools, transformations, properties, …

Application-Relevant Abstractions

- Language for interfacing with autotuning must convey concepts that are meaningful to the application programmer
- Wrong: unroll by 5
  - Though could be ok for performance expert, and some compilers already provide pragmas for specific transformations
- Right (maybe): Performance precious, typical loop count between 100 and 10000, even, not power of 2
- Middle ground: Apply unroll, align, SIMD transformations and tune
- We need work at developing higher-level, performance-oriented languages or language extensions
  - This would be the "good" future
  - Early steps include TCE, Orio, Spiral, …

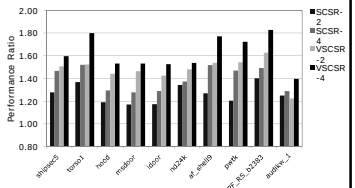
Better Algorithms and Data Structures

- Autotuning only offers the best performance with the given data structure and algorithm
  - That's a big constraint
- Processors include hardware to address performance challenges
  - "Vector" function units
  - Memory latency hiding/prefetch
  - Atomic update features for shared memory
  - Etc.

Sparse Matrix-Vector Multiply

Barriers to faster code

- "Standard" formats such as CSR do not meet requirements for prefetch or vectorization
- Modest changes to data structure enable both vectorization, prefetch, for 20-80% improvement on P7



Prefetch results in *Optimizing Sparse Data Structures for Matrix Vector Multiply*
http://hpc.sagepub.com/content/25/1/115

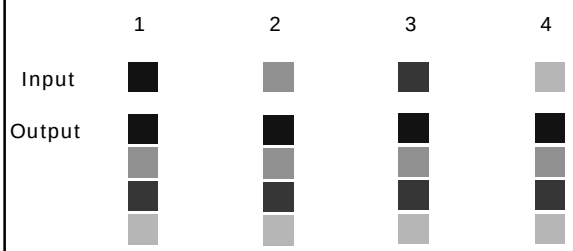
What Does This Mean For You?

- It is time to rethink data structures and algorithms to match the realities of memory architecture
  - We have results for x86 where the benefit is smaller but still significant
  - Better match of algorithms to prefetch hardware is necessary to overcome memory performance barriers
- Similar issues come up with heterogeneous processing elements (someone needs to *design* for memory motion and concurrent and nonblocking data motion)

Is It Communication Avoiding Or Minimum Solution Time?

- Example: non minimum collective algorithms
- Work of Paul Sack; see "Faster topology-aware collective algorithms through non-minimal communication", PPoPP 2012
- Lesson: minimum communication *need not be optimal*

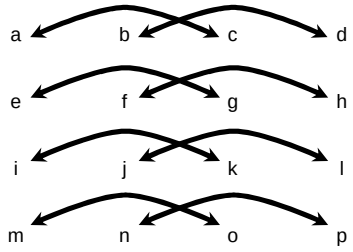
Allgather



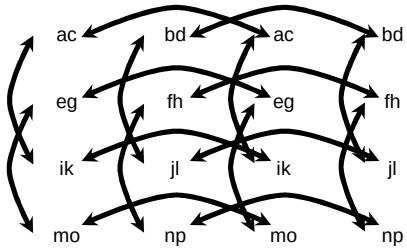
Problem: Recursive-doubling

- No congestion model:
  - $T=(\lg P)\alpha + n(P-1)\beta$
- Congestion on torus:
  - $T\approx(\lg P)\alpha + (5/24)nP^{4/3}\beta$
- Congestion on Clos network:
  - $T\approx(\lg P)\alpha + (nP/\mu)\beta$
- Solution approach: move smallest amounts of data the longest distance

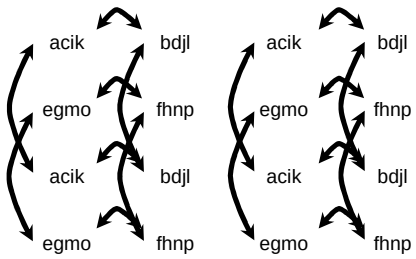
Allgather: recursive halving



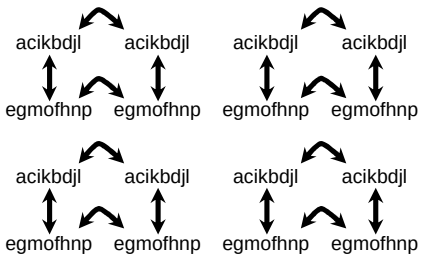
Allgather: recursive halving



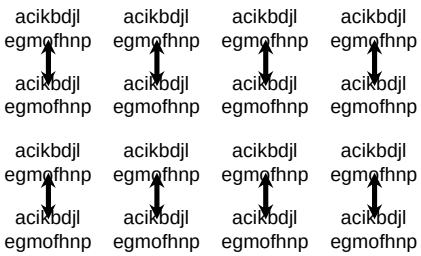
Allgather: recursive halving



Allgather: recursive halving



Allgather: recursive halving

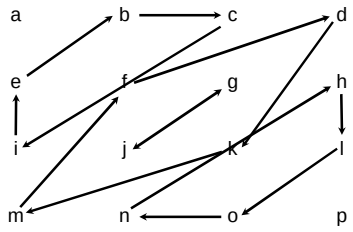


$T = (lg\ P)\alpha + (7/6)nP\beta$

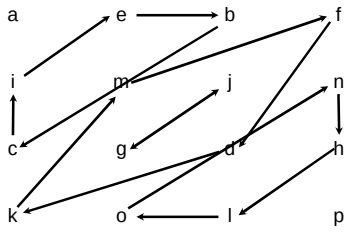
New Problem: Data Misordered

- Solution: shuffle input data
  - Could shuffle at end (redundant work; all processes shuffle)
  - Could use non-contiguous data moves
  - Shuffle data on network…

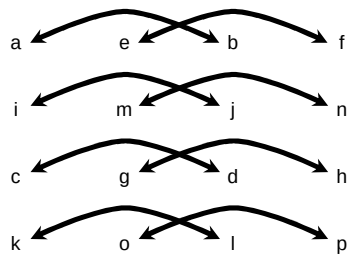
Solution: Input shuffle



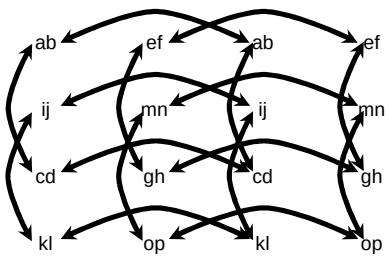
Solution: Input shuffle



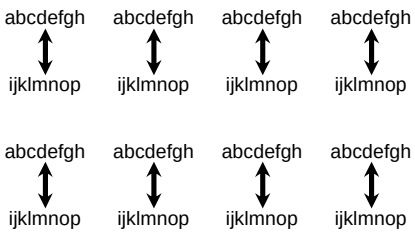
Solution: Input shuffle



Solution: Input shuffle



Solution: Input shuffle



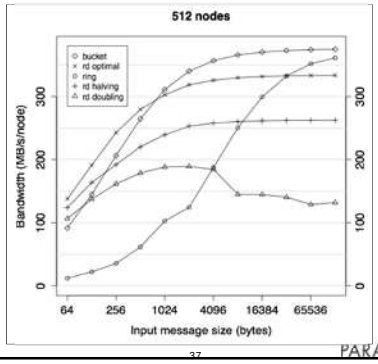
$T=(1+lgP)\,\alpha + (7/6)nP\beta$

$T\approx(lgP)\alpha + (7/6)nP\beta$

Evaluation: Intrepid BlueGene/P at ANL

- 40k-node system
  - Each is 4 x 850 MHz PowerPC 450
- 512+ nodes is 3d torus; fewer is 3d mesh
- xlc -O4
- 375 MB/s delivered per link
  - 7% penalty using all 6 links both ways

Allgather performance



Notes on Allgather

- Bucket algorithm (not described here) exploits multiple communication engines on BG
- *Analysis shows performance near optimal*
- Alternative to reorder data step is in memory move; analysis shows similar performance and measurements show reorder step faster on tested systems

Performance on a Node

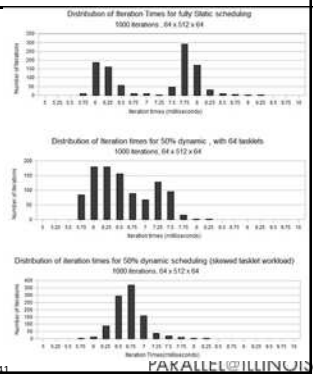
- Nodes are SMPs
  - You have this problem on anything (even laptops)
- Tuning issues include the usual
  - Getting good performance out of the compiler (often means adapting to the memory hierarchy)
- New (SMP) issues include
  - Sharing the SMP with other processes
  - Sharing the memory system

New (?) Wrinkle – Avoiding Jitter

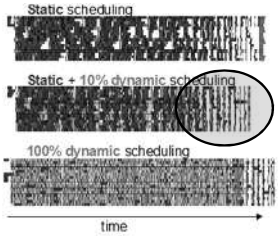
- Jitter here means the variation in time measured when running identical computations
  - Caused by other computations, e.g., an OS interrupt to handle a network event or runtime library servicing a communication or I/O request
- This problem is in some ways less serious on HPC platform, as the OS and runtime services are tuned to minimize impact
  - However, cannot be eliminated entirely

Sharing an SMP

- Having many cores available makes everyone think that they can use them to solve other problems ("no one would use all of them all of the time")
- However, compute-bound scientific calculations are often *written* as if all compute resources are owned by the application
- Such *static* scheduling leads to performance loss
- Pure dynamic scheduling adds overhead, but is better
- Careful mixed strategies are even better
- Thanks to Vivek Kale



Happy Medium Scheduling

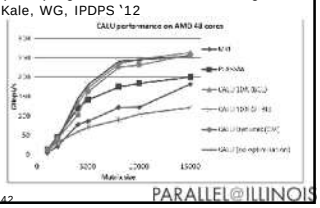


Scary Consequence: Static data decompositions *will not work at scale.*

Corollary: programming models with static task models *will not work at scale*

Performance irregularities introduce load-imbalance.
Pure dynamic has significant overhead; pure static too much imbalance.
Solution: combined static and dynamic scheduling

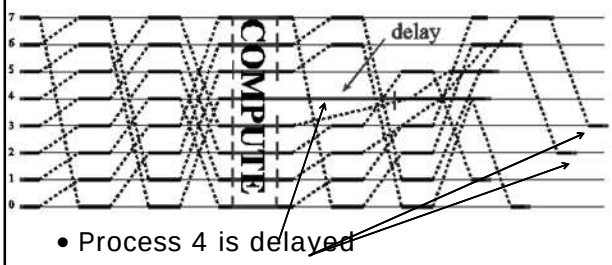
Communication Avoiding LU factorization (CALU) algorithm, S. Donfack, L .Grigori, V. Kale, WG, IPDPS '12



Synchronization and OS Noise

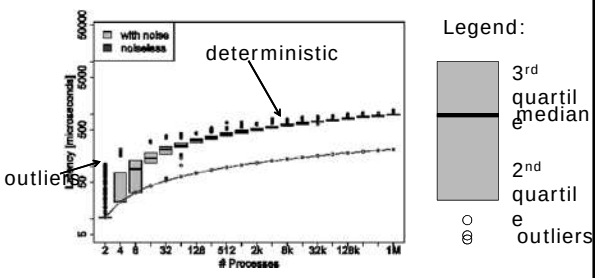
- "Characterizing the Influence of System Noise on Large-Scale Applications by Simulation," Torsten Hoefler, Timo Schneider, Andrew Lumsdaine
  - Best Paper, SC10
- Next 3 slides based on this talk…

A Noisy Example – Dissemination Barrier



- Process 4 is delayed
  - Noise propagates "*wildly*" (of course deterministic)

Single Collective Operations and Noise



- 1 Byte, Dissemination, regular noise, 1000 Hz, 100 µs

The problem is *blocking* operations

- Simple, data-parallel algorithms easy to reason about but inefficient
  - True for decades, but ignored (memory)
- One solution: fully asynchronous methods
  - Very attractive, yet efficiency is low and there are good reasons for that
  - Blocking can be due to fully collective (e.g., Allreduce) or neighbor communications (halo exchange)
  - Can we save methods that involve global, synchronizing operations?

Saving Allreduce

- One common suggestion is to avoid using Allreduce
  - But algorithms with dot products are among the best known
  - Can sometimes aggregate the data to reduce the number of separate Allreduce operations
  - But better is to reduce the impact of the synchronization by hiding the Allreduce behind other operations (in MPI, using MPI_Iallreduce)
- We can adapt CG to nonblocking Allreduce with some added floating point (but perhaps little time cost)

The Conjugate Gradient Algorithm

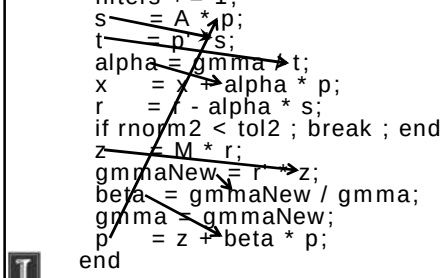
- While (not converged)

```
  niters += 1;
  s     = A * p;
  t     = p' *s;
  alpha = gmma / t;
  x     = x + alpha * p;
  r     = r - alpha * s;
  if rnorm2 < tol2 ; break ; end
  z     = M * r;
  gmmaNew = r' * z;
  beta  = gmmaNew / gmma;
  gmma = gmmaNew;
  p     = z + beta * p;
end
```

The Conjugate Gradient Algorithm

- While (not converged)

```
niters += 1;
s     = A * p;
t     = p' * s;
alpha = gmma / t;
x     = x + alpha * p;
r     = r - alpha * s;
if rnorm2 < tol2 ; break ; end
z     = M * r;
gmmaNew = r' * z;
beta  = gmmaNew / gmma;
gmma  = gmmaNew;
p     = z + beta * p;
end
```



CG Reconsidered

- By reordering operations, nonblocking dot products (MPI_Iallreduce in MPI-3) can be overlapped with other operations
- Trades extra local work for overlapped communication
  - On a pure floating point basis, the nonblocking version requires 2 more DAXPY operations
  - A closer analysis shows that some operations can be merged
- *More work does not imply more time*

What's Different at Peta/Exascale

- Performance Focus
  - Only a little – basically, the resource is expensive, so a premium placed on making good use of resource
  - Quite a bit – node is more complex, has more features that must be exploited
- Scalability
  - Solutions that work at 100-1000 way often inefficient at 100,000-way
  - Some algorithms scale well
    - Explicit time marching in 3D
  - Some don't
    - Direct implicit methods
  - Some scale well for a while
    - FFTs (communication volume in Alltoall)
  - Load balance, latency are critical issues
- Fault Tolerance becoming important
  - Now: Reduce time spent in checkpoints
  - Soon: Lightweight recovery from transient errors

Preparing for the Next Generation of HPC Systems

- Better use of existing resources
  - Performance-oriented programming
  - Dynamic management of resources at all levels
  - Embrace hybrid programming models (you have already if you use SSE/VSX/OpenMP/...)
- Focus on results
  - Adapt to available network bandwidth and latency
  - Exploit I/O capability (available space grew faster than processor performance!)
- Prepare for the future
  - Fault tolerance
  - Hybrid processor architectures
  - Latency tolerant algorithms
  - Data-driven systems

Recommended Reading

- Bit reversal on uniprocessors (Alan Karp, SIAM Review, 1996)
- Achieving high sustained performance in an unstructured mesh CFD application (W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, B. F. Smith, Proceedings of Supercomputing, 1999)
- Experimental Analysis of Algorithms (Catherine McGeoch, Notices of the American Mathematical Society, March 2001)
- Reflections on the Memory Wall (Sally McKee, ACM Conference on Computing Frontiers, 2004)

Thanks

- Torsten Hoefler
  - Performance modeling lead, Blue Waters; MPI datatype
- David Padua, Maria Garzaran, Saeed Maleki
  - Compiler vectorization
- Dahai Guo
  - Streamed format exploiting prefetch, vectorization, GPU
- Vivek Kale
  - SMP work partitioning
- Hormozd Gahvari
  - AMG application modeling
- Marc Snir and William Kramer
  - Performance model advocates
- Abhinav Bhatele
  - Process/node mapping
- Elena Caraba
  - Nonblocking Allreduce in CG
- Van Bui
  - Performance model-based evaluation of programming models
- Funding provided by:
  - Blue Waters project (State of Illinois and the University of Illinois)
  - Department of Energy, Office of Science
  - National Science Foundation