

再配布禁止

※ 本資料に記載されている内容の無断転載・複製を禁じます

「富岳」利用セミナー：ハンズオン

2025年2月版



登録施設利用促進機関 / 文科省委託事業「HPCIの運営」代表機関
一般財団法人 高度情報科学技術研究機構



■ 「富岳」利用セミナー（中級編）

- ◆ https://www.fugaku.r-ccs.riken.jp/docs/workshop_materials

- 第一部：単体性能の最適化手法
- 第二部：MPI・LLIO
- 第三部：単体性能の最適化手法 2

■ 利用手引書 利用およびジョブ実行編 1.46版

- ◆ https://www.fugaku.r-ccs.riken.jp/doc_root/ja/user_guides/use_latest/ (※1)

※1 最新版へのリンクになっています。

■ 利用手引書 言語開発環境編 1.31版

- ◆ https://www.Fugaku.r-ccs.riken.jp/doc_root/ja/user_guides/lang_latest/ (※1)

■ プログラミングガイド（プログラミング共通編）

- ◆ https://www.fugaku.r-ccs.riken.jp/doc_root/ja/programming_guides/Programming_common_part_Programming_Guide_JA.pdf

■ プログラミングガイド（プロセッサ編）

- ◆ https://www.fugaku.r-ccs.riken.jp/doc_root/ja/programming_guides/Processors_Programming_Guide_JA.pdf

■ MPI使用手引書(tcsds-1.2.40)

- ◆ https://www.fugaku.r-ccs.riken.jp/doc_root/ja/manuals/tcsds-1.2.40/lang/MPI/j2ul-2565-01z0.pdf

■ プログラミングガイド（IO編）

- ◆ https://www.fugaku.r-ccs.riken.jp/doc_root/ja/programming_guides/IO_part_Programming_Guide_JA.pdf

ハンズオンの構成



- Theme 1: 01_processor-core
 - 01_polynomial
 - 02_simd-f
- Theme 2: 02_cache
 - 01_latency
 - 02_mm-vs-mv
- Theme 3: 03_shared-memory
 - 01_bandwidth
 - 02_thread-affinity
 - 03_data-locality
- Theme 4: 04_pure-mpi
 - 01_pingpong
 - 02_mpingpong
 - 03_tofu-barrier
- Theme 5: 05_mpi-openmp
 - 01_vcoord
 - 02_thread-level
 - 03_hybrid
- Theme 6: 06_LLIO
 - 01_localtmp
 - 02_sharedtmp

本ハンズオンでは講義のみ

ファイルシステムの活用

単一のプロセッサコアの利用と関連

メモ

- 水色の枠で囲まれた内容は、詳細な事柄や付加的な情報を扱っています。
- タイトルに # がついているページは、発展的な内容を扱っています。

これらは、講義では飛ばしますが、質問は受け付けます。

複数ノードの利用と関連

準備：「富岳」と富士通コンパイラ

- 実習を進める上で必要となる知識をまとめたものです。
- 完全なレファレンスガイドではありません。
- 内容やデフォルトの設定等は今後変わる可能性があります。必ずマニュアルや富岳ウェブサイトの情報を確認するようにしてください。



■ 「富岳」計算ノードの構成

◆ システム仕様: https://www.fugaku.r-ccs.riken.jp/fugaku_information

◆ 総ノード数: 158,976

◆ ノード (=CPU + メモリ)

● CPU: A64FX (命令セットアーキテクチャー: ARMv8.2-A SVE (512-bit SIMD))

- ノード当たりのプロセッサコア数: 48
- クロック周波数: 通常モード: 2.0GHz、ブーストモード: 2.2GHz
 - SIO/GIO/BIOノードは常に 2.2GHz
- **ブーストモードのみの使用は禁止。エコモードとの組み合わせが必須。**
 - rscgrp=smallはデフォルトでブーストエコモードに移行の予定
- ノード当たりの理論演算性能
 - 倍精度、通常モード (freq=2000, eco_state=0)
3072Gflop/s = 2.0GHz x 2FMA x 8SIMD x 2FP x 48コア
 - 倍精度、ブーストエコモード (freq=2200, eco_state=2)
1690Gflop/s = 2.2GHz x 2FMA x 8SIMD x 1FP x 48コア

● 主記憶 (メインメモリ) : HBM2 (ノードあたり32GiB)

- ユーザ・プログラムにおける利用上限は経験上 ~ 28GiB
- 理論メモリ帯域幅: 1024 GB/s

◆ インターコネクト (ノード間の結合)

● Tofu interconnect D

- トポロジー: 6D mesh/torus

メモ

- ブーストエコモードの指定方法
#PJM -L "freq=2200, eco_state=2"
- 通常モードの指定方法
#PJM -L "freq=2000, eco_state=0"
- ブーストエコモードと通常モードの比較
02_cache/02_mm-vs-mv ([Appendix](#))



■ Fortran

- ◆ クロスコンパイラ: frtpx
- ◆ オウンコンパイラ: frt

■ C (clangモード)

- ◆ クロスコンパイラ: fccpx -Nclang
- ◆ オウンコンパイラ: fcc -Nclang

■ C++ (clangモード)

- ◆ クロスコンパイラ: FCCpx -Nclang
- ◆ オウンコンパイラ: FCC -Nclang

メモ

覚え方: 富士通提供のコマンド (コンパイラ・プロファイラ)について

- 「px」が末尾につく → ログインノードで実行
- 「px」が末尾につかない → 計算ノードで実行

富士通(FJ)以外のArm (aarch64) 用命令を生成できるコンパイラ

- 例: GNU, LLVM, Arm
- FJ以外を使用する場合の注意点 → ライブラリ (特にBLAS/LAPACK, OpenMP, MPI) の利用で配慮が必要 (なことが多い)

コンパイルコマンド（富士通コンパイラ）(2/3)



■ 補足1: クロスコンパイラ (cross) vs. オウンコンパイラ (own)

- ◆ クロスコンパイラの使用が基本

- ログインノードでの作業が可能

- ◆ クロスコンパイラでうまくいかないとき → オウンコンパイラを使用

- 例: python経由でC/C++コンパイラを利用するとき (Cython, Cython, pybind11など)
 - 計算ノード (Arm) 用のpythonを実行する必要があるため
 - 例: 共有ライブラリ(*.so)を作成するとき

■ 本ハンズオンではクロスコンパイラを利用

- ◆ 参考として、オウンコンパイラ用の設定ファイルも提供する

コンパイルコマンド（富士通コンパイラ）(3/3)



■ 補足2: C/C++におけるClangモード vs. Tradモード

- ◆ オプション “-Nclang” 付きでコンパイル → Clangモード
 - [Pros] プログラムの移植に適する、C++で有用なコンパイラ最適化手法（例：インライン展開）に適する
 - [Cons] 最適化機能（特にループ関連）が限定的
- オプション “-Nclang” なしでコンパイル → Tradモード
 - [Pros] 最適化機能（特にループ関連）が豊富
 - [Cons] （新しい言語規格を使用しているような）プログラムの移植に難点

■ 本講習会では、主にClangモードを取り上げる

- ◆ Tradモードのコンパイルオプションは
Fortranとほぼ同じであり、読み替え可能
- ◆ 性能分析の結果とプログラムの特性に基づき
使い分けることが重要
- ◆ ClangモードとTradモードを混合することも可能
(翻訳単位を分けることが必要)
- ◆ Clang - Tradの混合: C++のときは結合している
STLライブラリに注意すること

メモ

Clangモードが優勢な実習の例

- 01_processor-core/01_polynomial/cpp_ExpressionTemplate

Tradモードが優勢な実習の例

- 01_processor-core/01_polynomial/cpp
- 03_shared-memory/01_bandwidth/ctrad

ClangとTradの混合の実習の例

- 01_processor-core/02_simd-if/cpp_mix-trad

優先的に覚えるべきオプション: Fortran (1/3)



■ 最適化の促進

◆ -Kfast

- 03 + (各種最適化オプション)
 - A64FXで高速に動作することが期待されるバイナリを生成

◆ -Ksimd=auto

- SIMD処理を促進、-02で誘導 (-Kfastでも誘導)
- SIMD処理適用の“強さ”をコンパイラ任せで設定 (auto)
 - 強さの制御: 1, 2 → 1 < 2
(例: if文付きのループのSIMD処理は“2”で対応)
 - デフォルトのベクトル長 (まとめて処理する配列の要素数に相当) : 512 bit

◆ -Kswp

- ソフトウェアパイプラインングを促進、-02で誘導 (-Kfastでも誘導)
 - swp: 命令スケジューリングに関する最適化

◆ -Kswp_policy=auto

- swpの最適化のアルゴリズムをコンパイラ任せで設定 (auto)
 - アルゴリズムの種類: small, large

メモ

SIMDや命令スケジューリングに関する実習

- 01_processor-core/01_polynomial/fortran
- 01_processor-core/02_simd-if/fortran

SIMD処理のベクトル長 (VL) について

- SVE: 128-bitの倍数で指定可能。A64FXでは128/256/512が対応可能。
- Ksimd_reg_sizeでベクトル長 (固定 or 可変) の変更は可能



優先的に覚えるべきオプション: Fortran (2/3)

■ 診断レポートおよび詳細な翻訳メッセージの表示

- ◆ -Nlst=t -Koptmsg=2

■ OpenMPによるスレッド並列のアクティブ化

- ◆ -Kopenmp

- デフォルトではオフ

■ 富士通SSL2 (BLAS/LAPACK)の利用 (static link)

- ◆ -SSL2 (逐次版) あるいは -SSL2BLAMP (スレッド並列版)

■ 最適化の抑制

- ◆ 基本: 抑制したい最適化のオプションに "no" をつける

- 例: -Knosimd (SIMD処理の促進の抑制)

- オプションにより "no" の位置が異なるのでマニュアルを確認すること

➤ 例: -Kloop_blocking の場合は -Kloop_noblocking で抑制

- ◆ -Kfp_precision

- 浮動小数点演算で演算誤差が出る可能性のある最適化をまとめて抑制

- -Kfast,fp_precision のように指定

メモ

診断レポートの表示に関する実習

- 01_processor-core/01_polynomial/fortran
- 01_processor-core/02 SIMD-if/fortran

OpenMPに関する実習

- 03_shared-memory/00_stream
- 05_mpi-openmp/03_hybrid/fortran/thread_nt2

富士通SSL2(BLAS/LAPACK)に関する実習

- 02_cache/02_mm-vs-mv/fortran



■ 覚えておくことを推奨するオプション

◆ -Kocl

- ソースファイルに挿入された富士通コンパイラの最適化指示子(!OCL ...)を有効化する

◆ -Nalloc_assign

- Fortran2003規格以降の割付け代入 (allocatableな変数への代入(=)) の動作を有効化する
 - f90やf95ベースのコードでは割付け代入で性能が下がることがある
→ -Nnoalloc_assignか従来の配列要素間代入（例: A(:) = B(:)）に変更

◆ -Kswp_weak

- swpの最適化を制御, ループ回転数が変動 or 短い場合に性能向上が期待される

◆ -Kzfill

- 高速ストアを有効化, メモリアクセスに律速されるケースで性能向上が期待される

◆ -Knoomitfp

- 関数やサブルーチンの呼び出しにおいてフレームポインターレジスタを保証する
 - -Kfastでは-Komitfpが誘導 → プロファイラのコールグラフがうまく取れない可能性がある

◆ -x[サブルーチン・関数名1],[サブルーチン・関数名2],...

- 指定されたユーザ定義のサブルーチン/関数をインライン展開する
 - インライン展開でコンパイルの時間が長くなる傾向となるので注意



優先的に覚えるべきオプション: C/C++(Clangモード) (1/3)

■ 最適化の促進

◆ -Ofast

- O3 + (各種最適化オプション)
→ A64FXで高速に動作することが期待されるバイナリを生成

◆ -fvectorize

- SIMD処理を促進, -O2で誘導 (-Ofastでも誘導)

◆ -msve-vector-bits=[512|scalable]

- SIMD処理におけるベクトル長 (まとめて処理する配列要素数) を指定

◆ -ffj-swp

- ソフトウェアパイプラインニング (swp, 命令スケジューリングに関する最適化) を促進
- msve-vector-bits=512 (512-bit固定ベクトル長)との併用が前提

◆ -ffj-interleave-loop-insns

- SIMD処理が適用されたループでインターリーブ (命令スケジューリングに関する最適化と関連) を適用
- msve-vector-bits=scalable (可変ベクトル長; Vector-length agnostic)と-fvectorizeとの併用が前提

メモ

-ffj-***: 富士通で導入したコンパイルオプションを意味する
• 「-ffj-」がつかないオプション: LLVM (clang)と同じオプションを意味する

SIMDや命令スケジューリングに関する実習

- 01_processor-core/01_polynomial/cpp
- 01_processor-core/02_simd-if/cpp

可変ベクトル長 vs. 固定ベクトル長

- SVEでは128-bitの倍数でベクトル長を設定可能
- A64FXでは128-bit/256-bit/512-bitが利用可能
- msve-vector-bits=512にするとベクトル長は512-bitに固定



優先的に覚えるべきオプション: C/C++(Clangモード) (2/3)

■ 診断レポートおよび詳細な翻訳メッセージの表示

◆ `-ffj-lst=t -Rpass=".*" -Rpass-missed=".*" -Rpass-analysis=".*"`

- `".*"`: 検索条件として「任意のキーワードにマッチ」を意味
 - 正規表現による特定のキーワード指定も可能

■ OpenMPによるスレッド並列のアクティブ化

◆ `-fopenmp`

- デフォルトではオフ

■ 富士通SSL2 (BLAS/LAPACKを含む) の利用 (static link)

◆ `-SSL2` (逐次版) あるいは `-SSL2BLAMP` (スレッド並列版)

■ 最適化の抑制

◆ 基本: 抑制したい最適化のオプションに"no-"をつける

- 例: `-fno-vectorize`: (SIMD処理の促進の抑制)

◆ `-ffj-fp-precision:`

- 浮動小数点演算で演算誤差ができる可能性のある最適化をまとめて抑制

メモ

診断レポートの表示に関する実習

- `01_processor-core/01_polynomial/cpp`
- `01_processor-core/02_simd-if/cpp`

OpenMPに関する実習

- `03_shared-memory/00_stream`
- `05_mpi-openmp/03_hybrid/c/thread_nt2`

富士通SSL2(BLAS/LAPACK)に関する実習

- `02_cache/02_mm-vs-mv/c`



■ 覚えておくことを推奨するオプション

◆ -ffj-ocl

- ソースファイルに挿入された富士通コンパイラの最適化指示子 (`#pragma fj ...`) を有効化する
 - LLVMの最適化指示子は (`#pragma clang ...`) は常に有効 (オプション不要)

◆ -fstrict-aliasing

- データ型が異なる変数について、ポインタの参照先メモリ領域で重なり合いがない (エイリアスがない) ことを前提に最適化を促進
 - cf. 関数の仮引数でrestrict修飾子 (C) の指定も考えられる
C++の場合は `_restrict_` (GNU拡張) で対応する.

◆ -fslp-vectorize

- Superword-Level Parallelism vectorizationの促進
- `-msve-vector-bits=scalable`との併用が前提

◆ -fno-omit-frame-pointer

- 関数やサブルーチンの呼び出しにおいてフレームポインターレジスタを保証する
 - `-Ofast`では`-fomit-frame-pointer`が誘導 → プロファイラのコールグラフがうまく取れない可能性がある

◆ -ffj-zfill

- 高速ストアを有効化, メモリアクセスに律速されるケースで性能向上が期待される

MPI (Message Passing Interface) の利用



■ コンパイルおよびリンクのコマンド (クロスコンパイラ)

- ◆ Fortran: mpifrtpx
- ◆ C: mpifccpx -Nclang
- ◆ C++: mpiFCCpx -Nclang
 - “mpi”をコンパイルコマンドの先頭につける
→ MPI関連のヘッダーファイルやライブラリが自動的に設定される

メモ

- MPIに関する実習
- 04_pure-mpi
 - 05_mpi-openmp

■ プログラムの実行

◆ 典型的な実行コマンド:

```
mpiexec -np [プロセス数] --stdout-proc [ファイル名] --stderr-proc [ファイル名] [実行ファイル名]
```

◆ 富士通MPIでの実行時の注意

- ファイルのリダイレクション (<, >) はNG
→ コマンドラインオプション (--stdin, --std-proc, etc.) で指定
- プロセスごとに標準出力/エラー出力を生成する場合: --std-proc, --stdout-proc, --stderr-procを使用
- 全プロセスからの標準出力/エラー出力を1つのファイルに出力するオプション (--std, --stdout, --stderr) の使用は不可
- "mpi"付きコンパイルコマンドで生成した実行可能ファイル
→ 必ずmpiexecで実行 (たとえ1プロセスでも)



覚えておくとよい環境変数 (1/3)

- 環境変数: プログラムの実行時に設定することで、システム (OSなど) の挙動や環境を制御
 - ◆ 赤字を優先的に覚えるとよい

■ スレッド並列関連

◆ OpenMP規格

- **OMP_NUM_THREADS**: OpenMPで使うスレッド数 (A64FX: 最小で1, 最大で48)
- **OMP_STACKSIZE**: OpenMPで利用するスタックメモリ(private変数の領域)のサイズ
 - 例: 512M → 512MiB, 1G → 1GiB
- **OMP_WAIT_POLICY**: OpenMPのスレッドの待ち状態を制御 (PASSIVE, ACTIVE)
- **OMP_SCHEDULE**: OpenMPのschedule指示節でruntimeの場合のスケジューリングを制御
- **OMP_PLACES, OMP_PROC_BIND**: thread affinityの制御と関連

◆ 富士通固有

- **FLIB_BARRIER**: スレッド間におけるバリア同期の設定 (SOFT, HARD)
- **FLIB_CNTL_BARRIER_ERR**: バリア同期に関する診断メッセージの表示 (TRUE, FALSE)
 - 1スレッド実行時で診断メッセージがあるのである → "FALSE"でメッセージの抑制が可能



■ 富士通MPI関連

◆ 富士通MPIにおけるMCAパラメータ (OMPI_MCA_で始まるもの)

- **OMPI_MCA_mpi_print_stats**
 - MPI統計情報の出力を制御
 - 0(出力なし; デフォルト), 1(ランク0に平均的情報を集約), 2(ランクごとに出力), など
 - OMPI_MCA_mpi_print_stats_ranks: MPI統計情報を出力するランクを指定 (mpi_print_stats >= 2 で意味がある)
- **OMPI_MCA_plm_ple_memory_allocation_policy**
 - NUMAメモリポリシーの制御 (localalloc, interleave_allなど); デフォルト: localalloc
- **OMPI_MCA_plm_ple_numanode_assign_policy**
 - NUMAノードのコア割当ポリシーの制御 (share_cyclic, share_bindなど); デフォルト: share_cyclic
 - 普通は設定せずにジョブを流して問題ない (スケジューラに任せる)

◆ その他の環境変数

- **PLE_MPI_STD_EMPTYFILE**
 - プロセスごとの標準出力/エラー出力で, 空ファイル作成の有無を制御 ("on", "off")
 - 基本的に"off"の指定を推奨 (デフォルトで"off")

メモ

cf. MCA (Modular Component Architecture) parameter in OpenMPI
• <https://docs.open-mpi.org/en/v5.0.x/mca.html>
• <https://www.open-mpi.org/faq/?category=tuning>



■ ラージページ関連（「富岳」のHPC拡張機能）

- **XOS_MMM_L_HPAGE_TYPE**

- ラージページの使用の有無を制御 (hugetlbfs, none)

- **XOS_MMM_L_PAGING_POLICY**

- ページング方式の制御 (":"で区切って, 3つ指定; 例: demand:demand:prepage)

- 1番目: 静的データ領域, 2番目: スタック領域, 3番目: ヒープ領域

- **XOS_MMM_L_arena_lock_type**

- メモリ割当の制御 (0, 1)

- スレッド並列でメモリ獲得の競合がある場合 → "1"で性能向上の可能性

メモ

ページング

- 仮想メモリ方式で、物理メモリと仮想メモリを対応させること
- マッピングは固定長のブロック単位 (ページ)

ジョブ統計情報ファイル



- ジョブ統計情報はpjsubコマンドの-sオプションまたは-Sオプションの指定によりジョブ終了時に出力される。

- ◆ ジョブ統計情報を格納するデフォルトのファイル名： *(job_name).(job_id).stats*
 - 会話型ジョブの場合： *STDIN.(job_id).stats*

- ◆ 項目の説明はmanマニュアル pjstatsinfoから得られる

```
[_LNlogin] man pjstatsinfo
```

メモ

-sオプションを指定した時の出力は、-s オプションを指定したときの出力に加えて、ノードごとの情報が追加されたもの。

- ◆ ジョブ統計情報ファイルの中身（抜粋）

```
PJM CODE : 0 # ジョブ終了コード
REASON : -
PRO EXIT CODE : -
EPI EXIT CODE : -
ELAPSE TIME (LIMIT) : 00:10:00 (600)
ELAPSE TIME (USE) : 00:00:52 (52)
USER CPU TIME (USE) : 30420 ms
SYSTEM CPU TIME (USE) : 10420 ms
CPU TIME (TOTAL) : 40840 ms
MEMORY SIZE (LIMIT) : unlimited <DEFAULT>
MEMORY SIZE (ALLOC) : unlimited
MAX MEMORY SIZE (USE) : 695.5 MiB (729219072) # メモリー使用量
```



性能分析: 基本

■ タイマー挿入による区間ごとの時間計測

- ◆ 例: `clock_gettime`で経過時間を取得

- (必要に応じて) 計測時間をもとにハードウェアスペックを示す性能指標に変換

- Flop/s = (計測区間に含まれる+と*の数) / (測定した時間)

- メモリバンド幅 (bytes/s) = (計測区間に含まれる配列の要素数をバイト単位に変換) / (測定した時間)

```
#include <time.h>
#include <stdio.h>
double get_elp_time() { /*タイマールーチン */
    struct timespec tp;
    clock_gettime(CLOCK_REALTIME, &tp);
    return tp.tv_sec+(double)tp.tv_nsec*1.0e-9;
}

int main(void) {
    double elp1, elp2;
    elp1 = get_elp_time();
    経過時間を測定したい部分
    elp2 = get_elp_time();
    printf("Elapsed time (sec)=%13.4f\n", elp2-elp1);
}
```

メモ

CPU時間による計測の注意

- ・並列計算では使用プロセッサコア数によりCPU時間は増大
- ・よって、CPU時間の最大値 or CPU時間の分布を見るべき

Fortranのタイマー計測

- ・`system_clock`サブルーチンを使用
- ・Cで作成されたタイマーを呼び出す (`ISO_C_BINDING`を使用するとよい)

性能分析: プロファイラ (1/3)



■ 基本プロファイラ (fipp): ツールによる性能分析を試みる場合に優先的に使用することを推奨

◆ サンプリングによる性能情報の取得, (基本的に)再コンパイル不要

- 取得できる主な情報
 - ホットスポットとボトルネックの探索に使用する
 - 関数単位での負荷分布
 - ソースにおける行単位の負荷
 - コールグラフ (関数間の呼び出し関係)
 - プロセス/スレッドごとの負荷分布(ロードインバランス)
 - MPI通信関数に要する時間

◆ コマンドの例: 性能情報 (cpupa, mpi) とコールグラフ(call)を取得, 分析結果をテキスト形式 (-tttext)で出力

- 収集(計算ノード) : fipp -C -Icpupa,mpi,call -Puserfunc -10 \
-d [情報収集用ディレクトリ名] [分析するプログラムの実行コマンド]
- 分析(ログインノード): fipp -A -Icall,cpupa,nobalance,mpi,src -Tall -btotal \
-pinput=0,limit=0 -10 \
-o [分析結果の出力ファイル名] -tttext -d [情報収集用ディレクトリ名]
 - スレッドからの負荷の評価方法: “-btotal” (総和で評価), “-bmax” (最大値で評価)

メモ

サンプリングとは?

- OSの割り込み機能により, 一定間隔で関数などの使用状況を取得. プログラム実行中の平均的な負荷情報がわかる

fippが関連する実習

- 05_mpi-openmp/03_hybrid/c/thread_nt2
- 05_mpi-openmp/03_hybrid/fortran/thread_nt2

性能分析: プロファイラ (2/3)



■ 詳細プロファイラ (fapp): 計測区間のプログラム実行時のハードウェア使用状況をモニター

- 計測用関数のプログラムの挿入が必要 (再コンパイルが必要)
- fipp等で分析箇所 (=計測区間) が特定されている場合に使用を検討
 - 目的: ハードウェアの使用状況に基づき, 「最適化の効果が想定通りか」, 「最適化の余地があるか」を確認するため

◆ コマンドの例: イベントstatisticsでモニター, 分析結果をテキスト形式で出力

- 収集(計算ノード) : fapp -C -Icpupa,mpi -Hevent=statistics \
 -d [情報収集用ディレクトリ名] [分析するプログラムの実行コマンド]
- 分析(ログインノード): fapppx -A -Icpupa,mpi -pinput=0,limit=0 \
 -o [分析結果の出力ファイル名] -ttext -d [情報収集用ディレクトリ名]

◆ イベント (-Hevent) について

- statistics (or pa1): 基本的な統計情報を収集
 - flop/s, 単位時間あたりのメモリアクセス (GB/s), SIMD命令率, サイクルあたりの命令数 (IPC), etc.
 - MPI通信関数の使用状況 (コール回数, 転送データのメッセージ長)
- 他のイベント (pa2-pa15): CPU性能解析で使用

メモ

fappで使用する計測用関数

- fapp_start: 計測開始点に挿入
- fapp_stop: 計測終了点に挿入

fappが関連する実習

- 05_mpi-openmp/03_hybrid/c/sendrecv.fapp
- 05_mpi-openmp/03_hybrid/fortran/sendrecv.fapp

性能分析: プロファイラ (3/3)



■ CPU性能解析 (CPU Performance Analysis)

- ◆ 複数回fapp計測を繰り返して精密な分析を実行
 - 1回 (pa1): 簡易, 5回: 基本, 11回: 標準, 15回: 詳細
 - ノード内の性能情報を詳細に把握したい場合に使用する
 - MPI通信関数に関する情報の収集は「簡易」で対応すること (fapp -C -Hevent=statistics)
- ◆ どのようなときに検討するか?
 - 「簡易」では収集できない情報が最適化の分析で必要な場合
 - キャッシュのミス率, 演算器/メモリアクセス等の待ち時間, サイクルあたりで実行された命令 (instruction commit), etc.
 - ハードウェアのどの部分を使い切れているかを系統的に調べる上では有用
- ◆ 収集と分析方法
 - イベント指定 (-Hevent) を変えながら複数回fappを繰り返す
 - 分析情報をcsv形式の出力で保存する (-tcsv)
 - csvファイルを富士通提供エクセルファイルに読み込ませてレポートを得る
 - エクセルファイル: cpu_pa_report.xlsx

メモ

CPU性能解析が関連する実習

- 01_processor-core/01_polynomial/cpp.fapp
- 01_processor-core/01_polynomial/fortran.fapp
- 02_cache/02_mm-vs-mv/c.fapp
- 02_cache/02_mm-vs-mv/fortran.fapp

CPU性能解析を実行する上の注意

- 性能結果を安定化させるために, コアバインドを意識する必要がある (numactlコマンドを使用する必要がある)



コンパイルとリンク (1/2)

■ コンパイルとリンクの基本的な手順 (fccpxの場合で説明)

1. コンパイルを実行しソースプログラムを翻訳 (-c) → オブジェクトファイルを作成する

```
# オブジェクトsrc1.oを作成  
$ fccpx -Nclang [options] -c src1.c
```

2. 生成されたオブジェクトファイルと(必要なら) 外部ライブラリを全て結合 (リンク) → 実行可能ファイルを作成する

```
# src1.oとsrc2.oをリンクして実行可能ファイルmyprog.exeを作成.  
# 外部ライブラリとしてFFTW (libfftw3.a or libfftw3.so)を利用  
# (FFTWのインストール場所は[path]にあるという前提)  
$ fccpx -Nclang [options] src1.o src2.o -L[path] -lfftw3 -o myproc.exe
```

```
# src1.oとsrc2.oをリンクして実行可能ファイルmyprog.exeを作成.  
# 外部ライブラリとして富士通SSL2のBLAS/LAPACKを使用 (static link版)  
# コンパイラのオプションで対応可能  
$ fccpx -Nclang -SSL2 [other options] src1.o src2.o -o myproc.exe
```



コンパイルとリンク (2/2)

■ 実習では

- ◆ 一連の手続きをMakefileとしてまとめている

```
# 以下のコマンドでOK  
$ make -f [Makefile名] |& tee make.log
```

- ◆ あるいは上記コマンド群をまとめたスクリプトを提供している

```
# 自動実行版（課題によっては用意されていない）create_project.sh内にmake ... が記述されている。  
$ bash create_project.sh
```

ジョブ実行



- 作業は必ずデータ領域で行ってください。
- グループに割り当てられたデータ領域の確認

```
$ accountd -E
COLLECTDATE : 2025/01/02 0:34:56      unit[GiB]
USER : u1xyzw
*-----[GROUP]-----*
GROUP      VOLUME      LIMIT      USAGE      AVAILABLE      FILES      USE_RATE
hp240xxx   vol0001    5,120      1          5,119          1          0.0%
hp240xxx   vol0601    409,600    43,953    365,647    7,093,815    10.7%
/vol0601/data/hp240xxx
/vol0601/share/hp240xxx
*-----[USER]-----*
USER      VOLUME      LIMIT      USAGE      AVAILABLE      FILES      USE_RATE
u1xyzw    vol0601    20          1          19          59          0.0%
/vol0601/home/u1xyzw
```

グラフの描画



- spackにより管理・提供されているプログラムを使用して描画できる。

```
[_LNlogin]$ . /vol0004/apps/oss/spack/share/spack/setup-env.sh  
[_LNlogin]$ spack load py-pandas arch=linux-rhel8-cascadelake  
[_LNlogin]$ spack load py-matplotlib arch=linux-rhel8-cascadelake  
[_LNlogin]$ spack load gnuplot  
[_LNlogin]$ python 02viz.py stat.csv
```

setup-env.sh および
spack loadコマンドはログ
イン後1度だけ行えばよい。

- 本ハンズオン用に作成したコンテナファイル (draw_graph-x86_64.sif)を利用しても良い。

```
[_LNlogin]$ singularity shell -B $PWD:$PWD sif_file/draw_graph-x86_64.sif  
Singularity> cd 01_processor-core/01_polynomial/cpp/visualize  
Singularity> python3 02viz.py stat.csv
```

- 作成したグラフは Fugaku OnDemandで見ることが可能。

◆ x windowが利用できる環境では、 imagemagickを使って見ることも可能。

```
[_LNlogin]$ spack load imagemagick  
[_LNlogin]$ display out.png
```

Theme 1: Efficient use of a single processor core

対応する実習

- [01_processor-core/](#)

システムの概要: CPUとメモリ (per CMG)

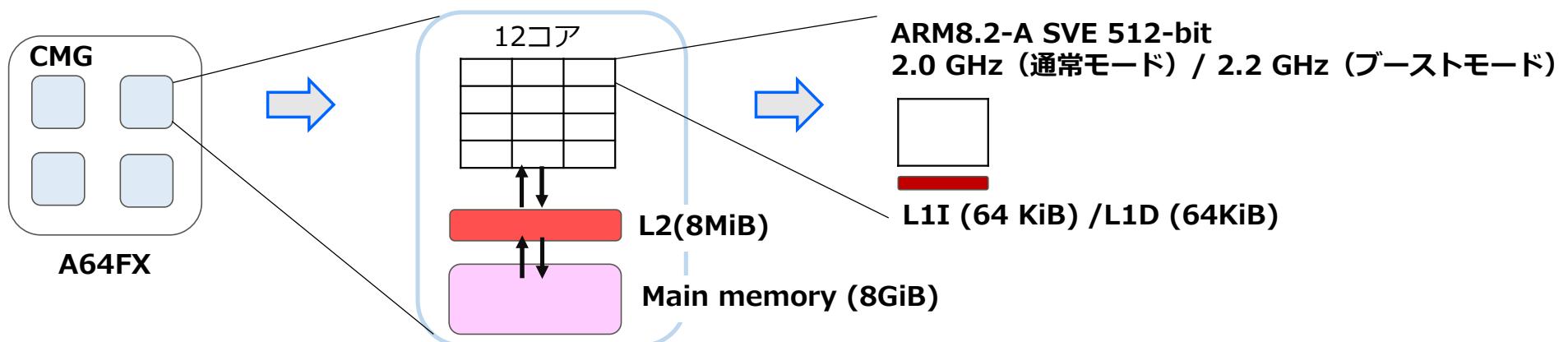


■ プロセッサコアの構成: 1個のCPUに48個の (ユーザが陽に使うことができる) コアを搭載

- ◆ フラットに48個が設置? → No → CPU内でnon-uniformに設置 (NUMA構成)
- ◆ コアと記憶装置 (キャッシュ, メモリ) が密に接続されたグループが4個設置
→ Core Memory Group (CMG)

■ CMGの構成

- ◆ 12個のプロセッサコアで8MiBのL2キャッシュ、および8GiBのメインメモリを共有
 - CMGが4個: L2キャッシュの総量 = $8\text{MiB} \times 4 = 32\text{MiB}$, メインメモリの総量 = $8\text{GiB} \times 4 = 32\text{GiB}$
- ◆ 1個のプロセッサコアに64KiBのL1I/L1Dキャッシュがつく (I=命令, D=データ)



Theme 1における着眼点



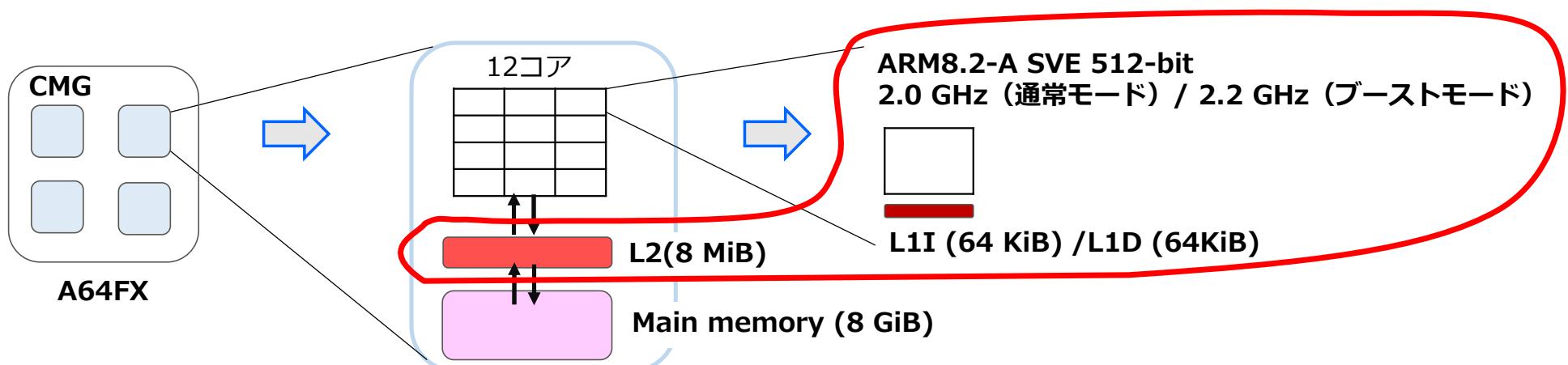
■ 単一のプロセッサコアに焦点

- ◆ メインメモリへのアクセスは(ほぼ)無い状況でプログラムを動かす
→ 演算 (+, *) を効率よく処理するパターン
 - 配列の要素数: L2 (あるいはL1D) キャッシュに載るようなデータサイズ

メモ
インターリープ
→ Tradモードのループストライピングに相当

■ 理解のポイント

- ◆ SIMD処理
 - 1個の命令でデータ(配列)をまとめて処理(レジスタヘロード/レジスタからストア, 演算)
- ◆ 命令スケジューリング(ソフトウェア・パイプライニング、ループインターリープ)
 - 命令を隙間なく並べて処理("待ち時間"を減らす)





多項式の評価 (01_polynomial) (1/7)[Fortran]

■ カーネル: 1次, 4次, 8次, 16次多項式をHorner's ruleで計算

- ◆ +と*の組み合わせ (積和演算のパターン) が繰り返す
 - 次数が高くなるほど演算量 (+と*の数) が増加: 演算量 = (次数) × 2
- ◆ 3種類の配列 (x , $c1$, $c2$) へのアクセスは連続アクセス

メモ

01_polynomialのパターン: EuroBenに含まれる9th degree polynomialに類似したカーネル

```
! Code fraction (mykernel.f90)
subroutine poly1 (ndim, x, c1, c2) ! 1st Degree

    do i = 1, ndim
        x(i) = c1(i)+x(i)*c2(i)
    end do
end subroutine poly1

subroutine poly4 (ndim, x, c1, c2) ! 4th Degree

    do i = 1, ndim
        x(i) = c1(i)+x(i)*(c2(i)+x(i)*(c1(i)+x(i)*(c2(i)+x(i)*c1(i) )))
    end do
end subroutine poly4
```



多項式の評価 (01_polynomial) (1/7) [C++]

■ カーネル: 1次, 4次, 8次, 16次多項式をHorner's ruleで計算

- ◆ +と*の組み合わせ (積和演算のパターン) が繰り返す
 - 次数が高くなるほど演算量 (+と*の数) が増加: 演算量 = (次数) × 2
- ◆ 3種類の配列 (x , c_0 , c_1)へのアクセスは連続アクセス

メモ

01_polynomialのパターン: EuroBenに含まれる9th degree polynomialに類似したカーネル

```
! Code fraction (mykernel.cc)
void poly1 (double * _restrict_cpp x, double * _restrict_cpp c0,
            double * _restrict_cpp c1, const int ndim)
{ // 1st Degree
    for( int i=0; i<ndim; ++i ) {
        x[i] = c0[i]+x[i]*c1[i] ;
    }
}
void poly4 (double * _restrict_cpp x, double * _restrict_cpp c0,
            double * _restrict_cpp c1, const int ndim)
{ // 4th Degree
    for( int i=0; i<ndim; ++i ) {
        x[i] = c0[i]+x[i]*(c1[i] +x[i]*(c0[i] +x[i]*(c1[i] +x[i]*(c0[i] )))) ;
    }
}
```

```
! cpp/mykernel.h
#define _restrict_cpp __restrict__
```



■ 設定

- ◆ データサイズ: (倍精度)配列サイズ = 10000
 - $3(\text{種類}) \times 10000 \times 8.0 \text{ B} = 234.4 \text{ KiB}$
→ L2キャッシュにおさまる (メモリアクセスを気にしなくてよい設定)
- ◆ 時間計測: 各カーネルを100000回繰り返し実行し, 経過時間を測定
(nrep in main.f90/main.cpp)
 - $[(\text{演算量}) \times (\text{nrep})] / (\text{経過時間})$ で flop/s を評価

■ ポイント

- ◆ 最適化オプション変更に対する診断レポート (lstファイル) の違いを確認する
- ◆ 最適化レベルの上昇による性能 (経過時間およびflop/s) の変化を確認する



多項式の評価 (01_polynomial) (3/7)

■ コンパイラの診断レポート(*.lst) [Fortran]

◆ ポイント:

“Loop-information”と(optimize)の記号を確認する

メモ

- line-no.: ソースコードの行数
- nest: ループのネスト(入れ子)の深さ
- optimize: 最適化の情報
 - 記号の前の数字: ループアンローリングの段数
 - 記号: 最適化の適用状況を表示(v: ベクトル化適用)

```
# mykernel.lst: config/Makefile.simdの場合
# -Kfast -Ksimd=auto -Knoswp
(line-no.)(nest)(optimize)
# On poly4 (4th Degree polynomial)
    <<< Loop-information Start >>>
    <<< [OPTIMIZATION]
    <<< SIMD(VL: 8)          ←要素数8でベクトル化
    <<< PREFETCH(HARD) Expected by
        compiler :
    <<<     c1, x, c2
    <<< Loop-information End >>>
24   1      2v           do i = 1, ndim
25   1      2v           x(i) = c1(i)      &
26   1                   +x(i)*(c2(i)+...
27   1                   )))                ←2段でアンロール
28   1      2v           end do          ベクトル化は適用(v)
```

```
# mykernel.lst: config/Makefile.swpの場合
# -Kfast -Ksimd=auto -Kswp -Kswp_policy=auto
(line-no.)(nest)(optimize)
# On poly4 (4th Degree polynomial)
    <<< Loop-information          ←(ループ回転数)>112 (=ITR)で
    <<< [OPTIMIZATION]          swp適用
    <<< SIMD(VL: 8)          ←要素数8でベクトル化
    <<< SOFTWARE PIPELINING(IPC: 1.61,
        ITR: 112, MVE: 4, POL: S)
    <<< PREFETCH(HARD) Expected by
        compiler :
    <<<     x, c2, c1
    <<< Loop-information End >>>
24   1      2v           do i = 1, ndim
25   1      2v           x(i) = c1(i)      &
26   1                   +x(i)*(c2(i)+...
27   1                   )))                ←2段でアンロール
28   1      2v           end do          ベクトル化は適用(v)
```

多項式の評価 (01_polynomial) (4/7)

■ コンパイラの診断レポート (*.lst) [C++]

◆ ポイント:

“Loop-information”と(optimize)の記号を確認する

メモ

- line-no.: ソースコードの行数
- nest: ループのネスト（入れ子）の深さ
- optimize: 最適化の情報
 - 記号の前の数字: ループアンローリングの段数
 - 記号: 最適化の適用状況を表示 (v: ベクトル化適用)

```
# mykernel.lst: config/Makefile.clangの場合
# -Ofast -fvectorize -msve-vector-bits=scalable
(line-no.)(optimize)
# On poly8 (8th Degree polynomial)
```

- 要素数2でベクトル化
(可変長)
- インターリーブ未適用
(カウント=1)

```
24     v
25
26
...
28
29 }
```

for(int i=0; i<nndim; i++) {
 x[i] = c0[i]
 +x[i]*(c1[i] ...
))))))));

アンロール無し
ベクトル化は適用 (v)

```
# mykernel.lst: config/Makefile.clang.512.swpの場合
# -Ofast -fvectorize -msve-vector-bits=512 -ffj-swp
(line-no.)(optimize)
# On poly8 (8th Degree polynomial)
```

- 要素数8でベクトル化
- インターリーブ未適用
(カウント=1)

```
24     4v
25
26
...
28
29 }
```

for(int i=0; i<nndim; i++) {
 x[i] = c0[i]
 +x[i]*(c1[i] ...
))))))));

4段でアンロール
ベクトル化は適用 (v)



多項式の評価 (01_polynomial) (5/7)

■ ログ (logfile) の出力例

◆ Elapsed: 短いほど高性能

◆ Giga FLOPs: 高いほど高性能

- 演算 (+, *) の数が多い状況でFLOPsが上昇
(経過時間が大きく増大しない)

→ 演算を効率よく処理できている証拠

➤ 注意: メモリアクセスが支配的でない状況であることが
先駆的に判明しているためにできる考察

→ 通常のプログラムでは律速を見極めた上で議論が必要

Number of repetitions :	100000	経過時間 (sec.)
Number of elements :	10000	
<hr/>		
[run] 1 FMA		
...		
Elapsed (sec.)	: 0.3520	
Giga FLOPs	: 5.6818	
...		
[run] 4 FMA		
...		
Elapsed (sec.)	: 0.3490	
Giga FLOPs	: 22.9226	
...		
[run] 8 FMA		
...		
Elapsed (sec.)	: 0.3940	
Giga FLOPs	: 40.6091	
...		
[run] 16 FMA		
...		
Elapsed (sec.)	: 0.8730	
Giga FLOPs	: 36.6552	

多項式の次数
(=積和演算の数)

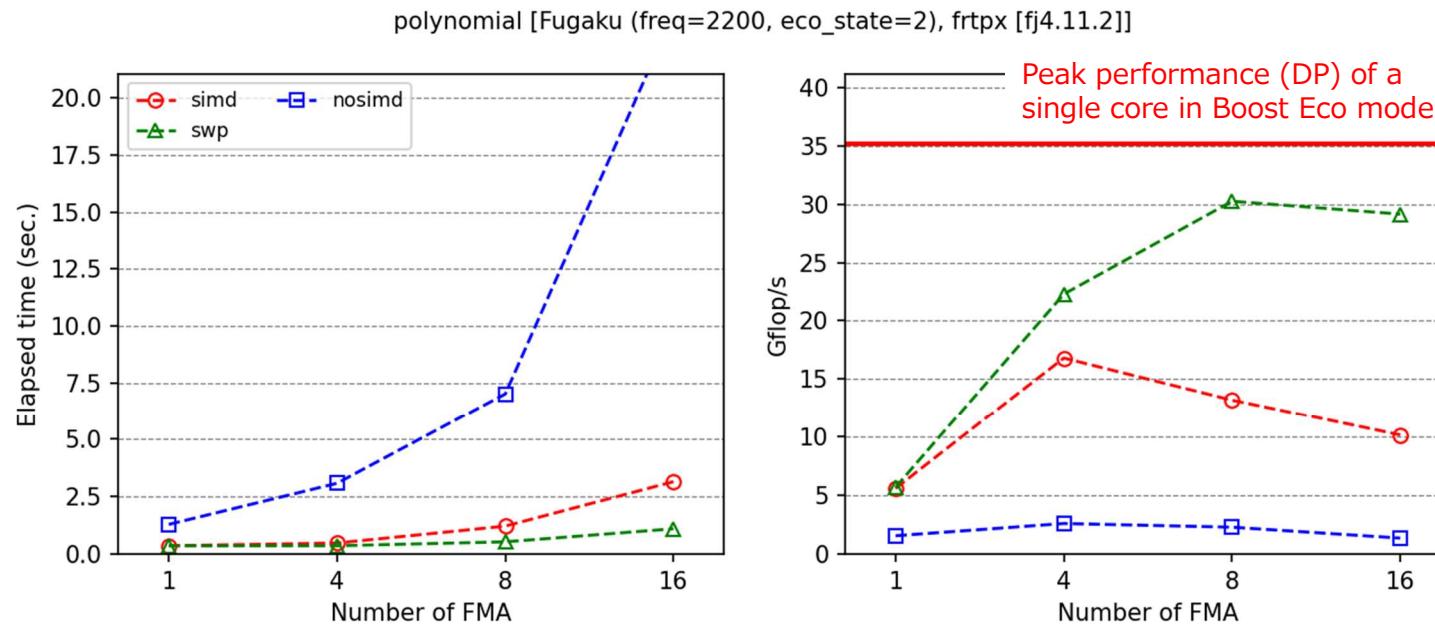
Gflop/s

演算量: 大

多項式の評価 (01_polynomial) (6/7)

■ 結果 [Fortran]

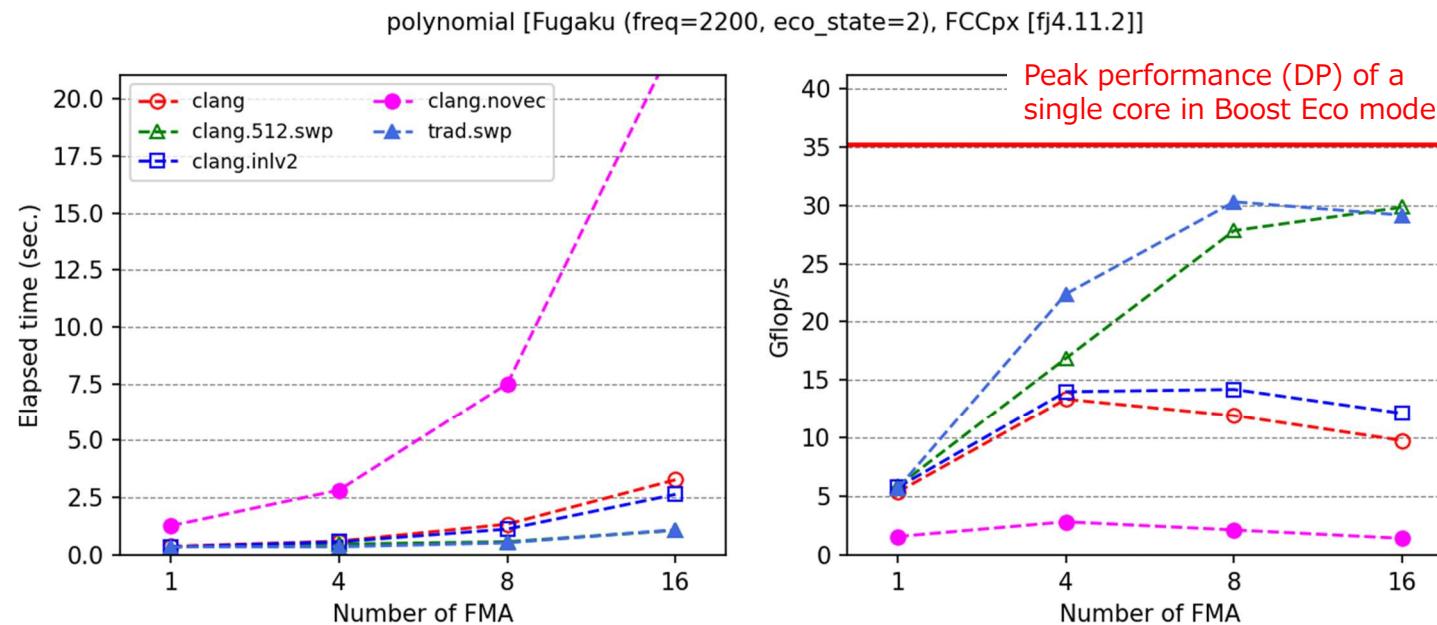
- ◆ ベクトル化未適用 ([nosimd](#)) → 大きく性能が落ちる
- ◆ 4FMA以降: ソフトウェアパイプラインニング ([swp](#)) が大きなimpact
→ 命令スケジューリングが大事
- ◆ 8FMA + swp +simd: 理論ピーク性能 (DP, a single core) の85%程度



多項式の評価 (01_polynomial) (7/7)

■ 結果 [C++]

- ◆ ベクトル化未適用 (`clang.novec`) → 大きく性能が落ちる
- ◆ 8FMA以降: ソフトウェアパイプラインング (`swp`) が大きなimpact → 命令スケジューリングが大事
 - cf. 可変長+インターリーヴ (`clang.inlv2`) の効きがよくない
→ このカーネルでは, 512-bit固定長+swp (`clang.512.swp`) が適切
- ◆ 16FMA + 512-bit + swp +simd: 理論ピーク性能 (DP, a single core) の85%程度
 - Tradモード+swp (`trad.swp`) (~Fortran) 相当の性能までcatch up



ハンズオンの概要



■ 01_processor-core/01_polynomialに取り組んでください。

- ◆ 最初に Readme.md に目を通してください。

■ Exercise A

- ◆ E1: “SIMD-ON” と “SIMD-OFF” の時のMakefileの違いを確認する。
- ◆ E2: 富士通コンパイラによる最適化情報(.lstファイル)を出力するオプションを確認する。
- ◆ E3: 最適化オプションによる最適化の違いを確認する(mykernel.lstを比較する)。
- ◆ E4: “SIMD-ON”と“SIMD-OFF”での性能を比較する。
- ◆ E5: 命令スケジューリングに関するコンパイルオプションを使用したときの性能を確認する。

■ Exercise Aが終わったら以下のどれかに取り組んでください。

- ◆ Exercise B
- ◆ 01_processor-core/02_simd-if (**Appendix**)
- ◆ A64FX Tuning Documentに目を通す。
https://github.com/RIKEN-RCCS/A64FX_Tuning_Documents

Theme 2: Use of cache in memory hierarchy

対応する実習

- 02_cahce/



Theme 2における着眼点

■ キャッシュ (と単一のプロセッサコア)に着目

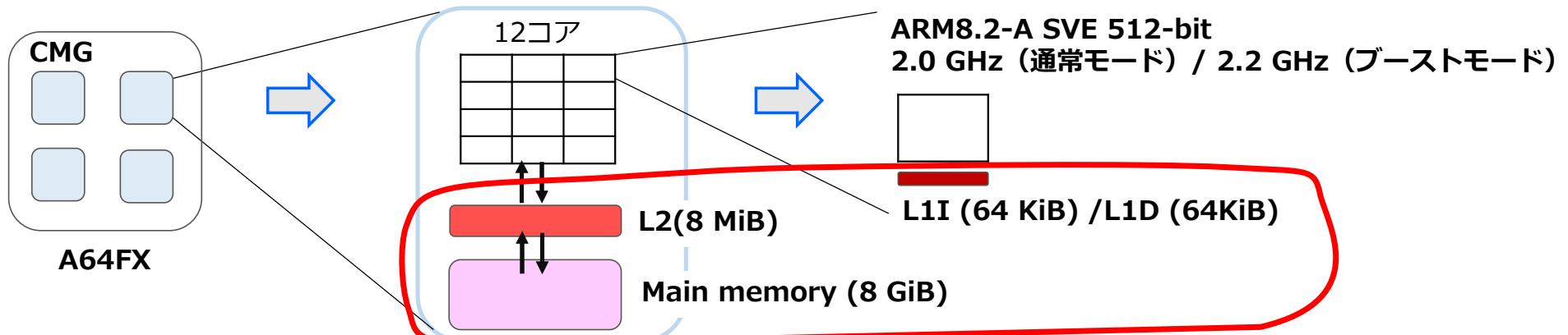
- ◆ メインメモリのアクセスが発生する状況でプログラムを動かす
→ キャッシュの特性とその有効利用のパターン
 - 配列の要素数: 最大で, L2キャッシュ/CMGを超えるデータサイズ

■ 理解のポイント

- ◆ レイテンシ (応答時間, 遅延時間)
 - デバイスの立ち上げに必ず要する時間 (キャッシュ: データ転送が始まるまでかかる時間)

メモ

- 行列行列積 (DGEMM)
 - キャッシュチューニング (ブロッキング, タイリング) が効果的な代表的なパターン
 - 富士通がA64FX向けに最適化された関数(ライブラリ)を提供 (優先的に利用すべき)



Quick review: キャッシュについて



■ キャッシュとは

- ◆ メインメモリとレジスタの中間に位置する記憶装置
 - A64FX: L1キャッシュ (コア固有), L2キャッシュ (コア共有) の2段構成

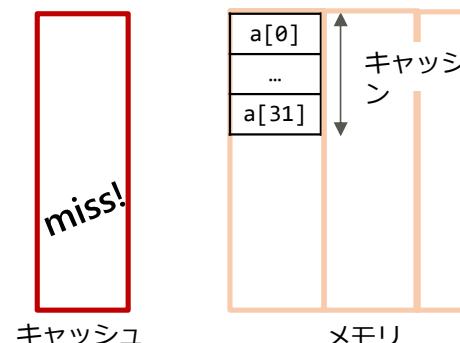
■ キャッシュ-メモリ間のデータ移動

◆ キャッシュラインの単位で移動

- A64FXのラインサイズ (line sizes): 256 bytes (倍精度配列の32要素)
- 同一キャッシュライン上のデータ: 将来の計算で使われることが期待される
- キャッシュミスのペナルティ: データ転送の時間 (バンド幅 × サイズ) + レイテンシ (デバイスの応答時間)

```
double a[N];
for (i=0; i<N; ++i)
    a[i] = a[i] + 1.0;
```

i=0: a[0]が必要→キャッシュを探索
→無い場合: メモリからデータ移動



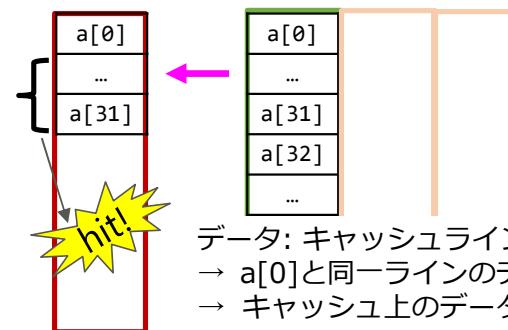
メモ

プリフェッチ (prefetch)

- キャッシュのレイテンシ隠蔽のために利用される機能
 - ・ 使われる予定の配列要素を先出しで読み取る
 - ・ *.1stファイルに関連メッセージが表示されることがある
- 基本: コンパイラ任せでよい
 - ・ 必要なら: 最適化指示子, prefetch用の関数 (`_builtin_prefetch` in GNU) で制御 (advanced)

キャッシュのway数 (連想度): ラインサイズと同様にキャッシュの構成を特徴づける量。A64FXの場合:

- L1D: 4-way (set-associative)
→ 1wayあたり16KiB ($16 \times 4 = 64$ KiB)
- L2/CMG: 16-way (set-associative)
→ 1wayあたり512KiB ($512 \times 16 = 8 \times 1024$ KiB)



LMbenchによるレイテンシ計測 (01_latency) (1/3)



■ LMbench: (UNIX/POSIXベースの)計算機について性能が計測可能なシンプルなベンチマーク
(<https://lmbench.sourceforge.net/>)

◆ lat_mem_rd: “memory read” に関するベンチマーク → 記憶装置のレイテンシを計測可能

- メモリ (アドレス) へのアクセスを何回も繰り返す
- アクセスの仕方 (ストライド) を変更して計測する
- ストライドに応じたレイテンシの傾向
→ ラインサイズの見積りに使える

■ 設定

◆ メモリサイズ (Range): 最大で256MiBまで変動

- L2/CMG(=8MiB), L2全体 (=32MiB)を超えるサイズまで動かす

◆ ストライド (メモリへのアクセスの距離): 64, 128, 256, 512, 1024 bytesと変化

- A64FXのラインサイズ (256 bytes) を含むように設定

➤ ラインサイズより短いストライドの場合: (キャッシュにアクセスする可能性が高いため) レイテンシが低くなることが期待

■ ポイント

◆ メモリサイズに応じた計測時間 (レイテンシ) の挙動を観察

- 特にキャッシュのレベルが変わる点に注目

メモ

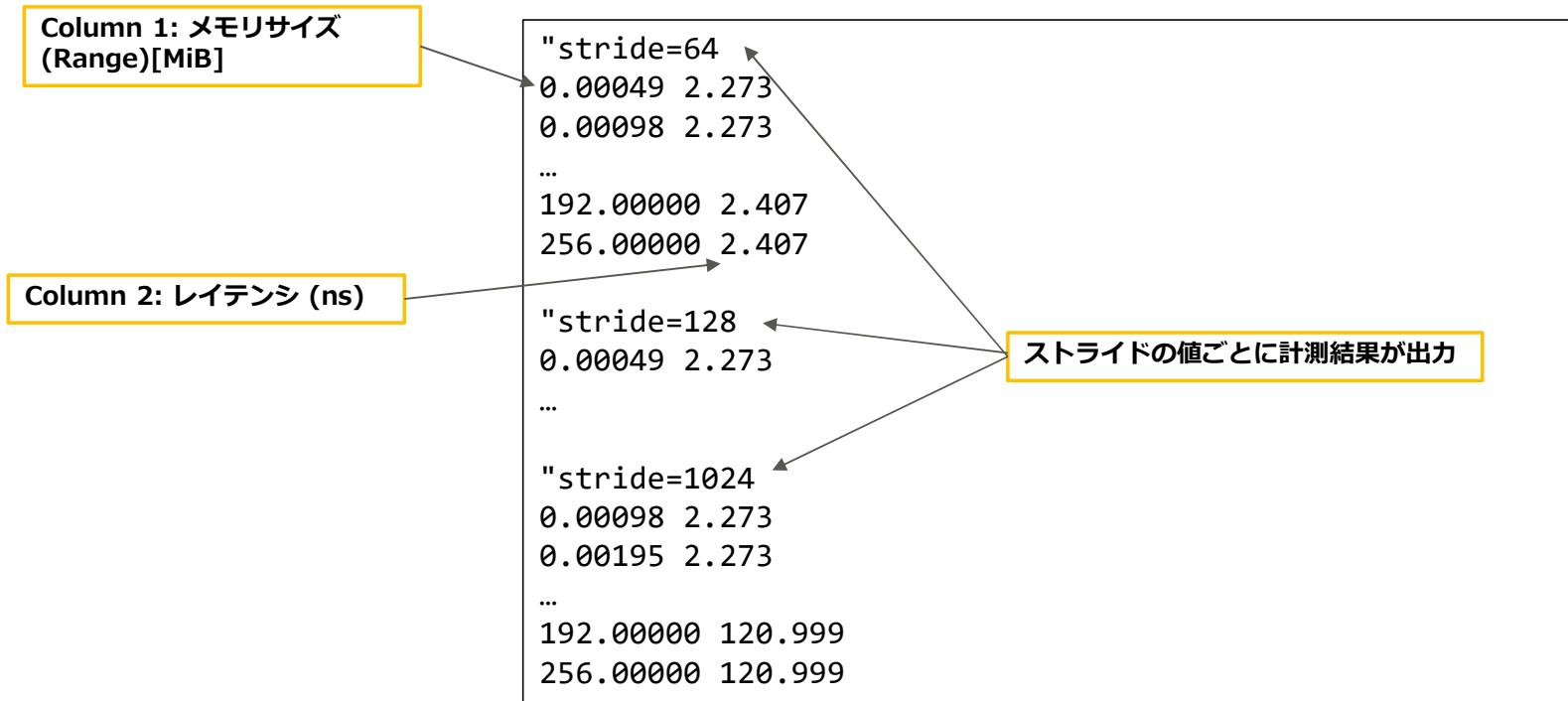
コンパイル上の注意

- OS (UNIX/Linux)側のライブラリを利用 → OSの仕様に鋭敏
- Red Hatの場合: RHEL8以降でrpc.hが見つからない可能性大
 - libtirpc-develをインストールする必要がある
- 00_lmbench/: 「富岳」における構築手順を同梱
(00_lmbench/Readme.mdを参照)

LMbenchによるレイテンシ計測 (01_latency) (2/3)



- ログの出力例 (ジョブスクリプトのLOGFILEで指定したファイル名)



LMbenchによるレイテンシ計測 (01_latency) (3/3)



■ 結果

◆ 全てストライドについて

- (Range) < L1D → レイテンシは同一 ~ 2.3ns

◆ (ストライド) > 256

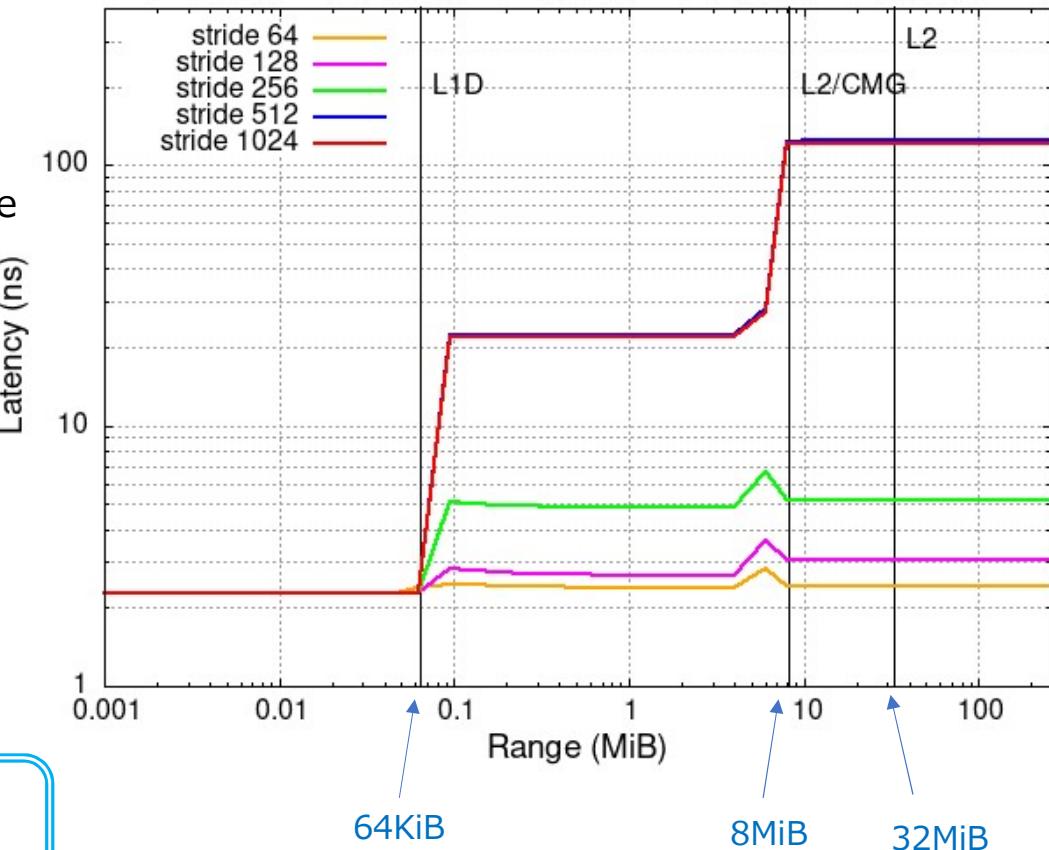
- (Range) > L1D でレイテンシのabrupt change
→ L2の領域; レイテンシ~22ns
➤ Recall: A64FXのラインサイズは256bytes

- (Range) > L2/CMGで再びabrupt change,
その後は一定 (L2のフルサイズを超えてなお)
→ (レイテンシについて言えば) メモリアクセス
の領域と見なせる; レイテンシ~120ns

メモ

- A64FXのクロックサイクル時間 ~0.46 ns (1/2.2GHz)
- ハードウェアスペックとしてのキャッシュのレイテンシはA64FXのドキュメントで確認すること

LMbench:lat-mem-rd (-P 1 256) at Fugaku (freq=2200, eco_state=2) [fj4.11.2]



Theme 3: Shared-memory parallelism with OpenMP in NUMA architecture

対応する実習

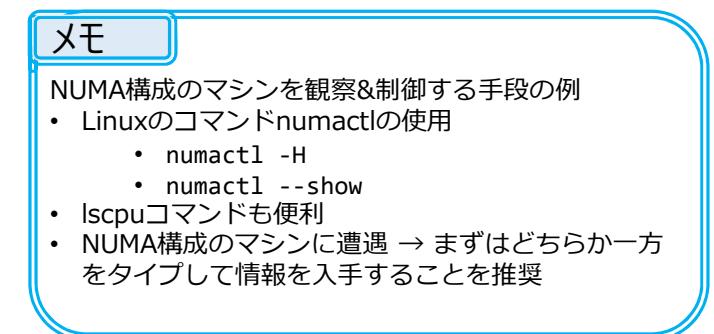
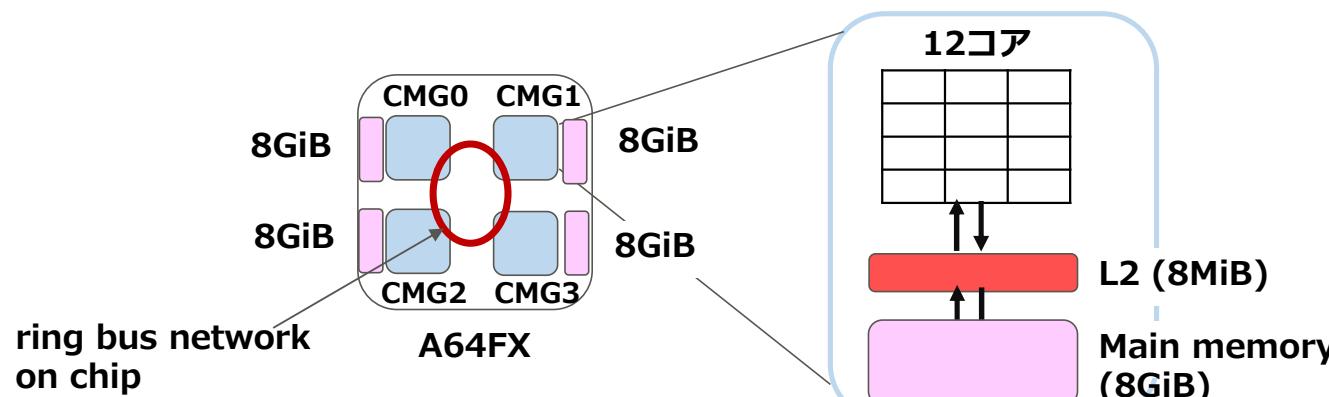
- [03_shared-memory/](#)



システムの概要: 共有メモリ

■ CMG間でメモリは共有

- ◆ リングバスから成るon-chip ネットワーク (NoC) でCMG間は接続
 - あるCMGに属するプロセッサコアから別のCMGの近くに置かれているメモリにアクセス可能
→ 「単一のA64FXは32GiBの共有メモリ型計算機」と見なせる
 - 重要な注意: 異なるCMG間でメモリアクセスは「距離」が遠い
→ NUMA (Non-Uniform Memory Access) 構成に特有の状況
 - A64FXでの考え方 (の例)
 - (スレッド数) ≤ 12 (=CMGのコア数) → CMG内にスレッドを配置, CMGを「ノード」と見なしMPI通信 (ハイブリット並列)
 - (スレッド数) > 12 (=CMGのコア数) → case-by-caseの対応が必要 (48スレッドまでの利用を考えるほうが設定はシンプルになる)





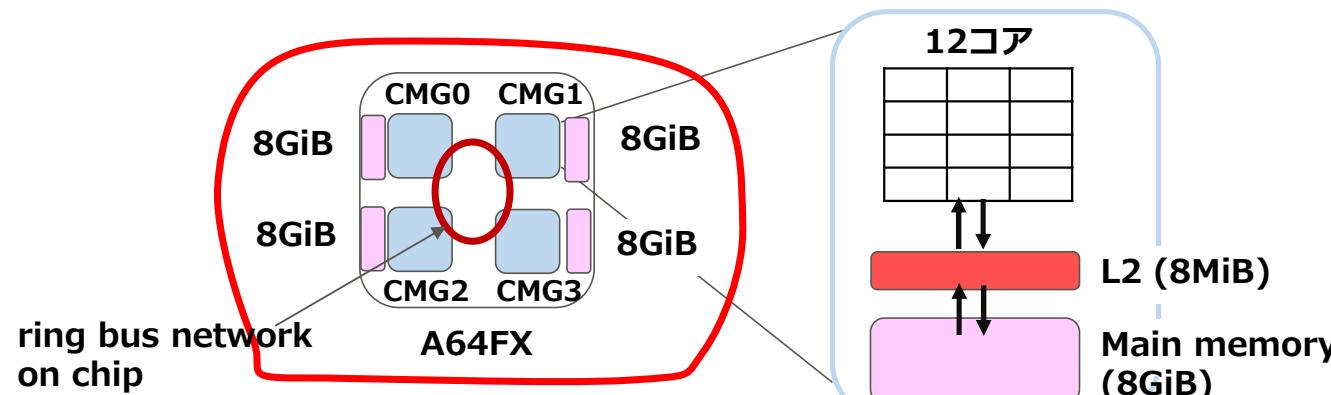
Theme 3における着眼点

■ メモリアクセスとマルチコアに着目

- ◆ メインメモリのアクセスが発生する状況でプログラムを動かす
→ メモリアクセスの特性とスレッド並列の性能を調べるパターン
 - 配列の要素数: L2キャッシュを大きく超えるようなデータサイズ
 - 使用するプロセッサコアの数: 最小で1, 最大で48 (共有メモリ並列)

■ 理解のポイント

- ◆ メモリバンド幅
 - 単位時間あたりのメモリからのデータ転送量; 複数コアでメモリにアクセスすることで増加する
- ◆ NUMA構成におけるスレッド並列
 - Thread affinity: スレッドの配置 (プロセッサコアとスレッドの結びつき(binding)) を制御
 - Data locality: データのCMG (or NUMA node)への配置に関する考慮点





Quick review: Thread affinityについて (1/2)

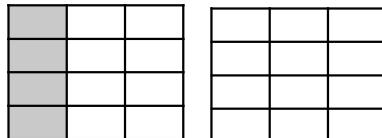
■ Thread affinityとは

- ◆ マルチコアの計算機においてスレッドのプロセッサコアに対する配置を定めること
(コアバインディング)

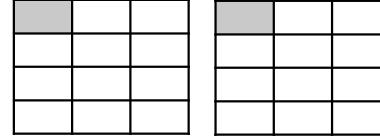
- 例: できるだけスレッド0の近く配置 (OMP_PROC_BIND=close)
- 例: 均等に配置 (OMP_PROC_BIND=spread)
- 例: コアIDと陽に結びつけて配置 (GOMP_CPU_AFFINITY="12-15")
 - A64FXの場合: ユーザが使うプロセッサコアのIDは12から59 (numactl -Hで確認可能)

プロセッサコアへの配置の例 (イメージ): スレッド数は4とする

OMP_PROC_BIND=close
(OMP_PLACES=cores)



OMP_PROC_BIND=spread
(OMP_PLACES=cores)



GOMP_CPU_AFFINITY="12-15"

12	16	20
13	17	21
14	18	22
15	19	23

GOMP_CPU_AFFINITY="12-48:12"

12	16	20
13	17	21
14	18	22
15	19	23

36	40	44
37	41	45
38	42	46
39	43	47

36	40	44
37	41	45
38	42	46
39	43	47



Quick review: Thread affinityについて (2/2)

■ A64FXにおけるaffinityの制御

◆ FLIB_BARRIER=HARDの場合 (ハードウェアバリアを有効)

- システム側で自動的に設定 (ユーザが制御しない)
 - 12スレッド以内かつプログラムの実行を通してスレッド数を変更しない場合に使用を検討

◆ FLIB_BARRIER=SOFTの場合 (ハードウェアバリアを非有効)

→ 環境変数による設定をおこなう

- 利用する環境変数 (どちらか一方を使うこと)
 - OMP_PLACESとOMP_PROC_BIND: コアIDを陽に指定する必要はない (OpenMP規格の環境変数)
 - GOMP_CPU_AFFINITY : コアIDを陽に設定; コンパイラ依存の環境変数 (富士通コンパイラでは利用可能) (advanced)

◆ 本講習会

- OMP_PLACESとOMP_PROC_BINDに基づく方法を使用 (コアIDを陽に指定しない)
- A64FXの場合: OMP_PLACES=coresでよい → 1プロセッサコアで1スレッドを結びつけるシステム
- OMP_PROC_BIND: closeかspreadを指定

メモ

ハードウェアバリアの効果: OpenMPの各種構文のオーバヘッド計測ベンチマークで確認が可能
(e.g., EPCC OpenMP microbenchmark suite)

Affinity制御の環境変数

- OpenMP規格以外: 利用の可否はコンパイラに依存
- GOMP_CPU_AFFINITY: 富士通コンパイラでLLVMのruntime (-Nlibompを指定)を利用するときに利用可能
 - GNUコンパイラ (libgomp)でも利用可能
 - KMP_AFFINITY: Intelコンパイラなどで利用される

Quick review: Data localityについて



■ Data localityとは

- ◆複数のNUMAノードがある計算機においてデータをスレッドがアクセスしやすいようにメモリに置くこと
 - A64FX: NUMAノードはCMGに対応
- ◆スレッドが属するNUMAノードに設置されたメモリにデータを置くことが好ましい

■ Linuxの場合: 基本は, first touch policy (ページングと関連) を満足するようにコーディングする

- ◆例: 配列の初期化をOpenMPのparallel領域で実施する

■ 「富岳」利用時の注意点

- ◆ページングのポリシーが2種類
 - demand: 配列に値が与えられたときにページを確保 (通常のLinuxの挙動)
 - prepage: あらかじめページを確保 (配列に値が入るまえにページが確保される可能性がある)
 - first touch policyを満足する場合でもprepageではdata localityが満足されない可能性がある
 - 12スレッド以上の高スレッドかつメモリ律速のカーネルでは注意が必要
- ◆ページングのポリシーは環境変数で制御可能: XOS_MMM_L_PAGING_POLICY
 - 本講習会: 03_data-localityを除きXOS_MMM_L_PAGING_POLICY=demand:demand:prepageに設定する

メモ

ページング

- 仮想メモリ方式で、物理メモリと仮想メモリを対応させること
- マッピングは固定長のブロック単位 (ページ)



STREAMによるメモリバンド幅の測定 (01_bandwidth) (1/5) [Fortran]

- STREAM: メモリからのデータ転送速度 (実効的なメモリバンド幅)を計測するベンチマーク
(<https://www.cs.virginia.edu/stream>)

- ◆ 4個のカーネル (Copy, Scale, Add, Triad)から構成

- 本講習会ではTriadに着目

- 浮動小数点演算の数: 2 per loop
 - メモリロード (=の右辺): 2*8bytes per loop,
メモリストア (=の左辺): 1*8bytes per loop

- ◆ 連続的なメモリアクセスのパターンを有するカーネルを計測

- 実効的なメモリバンド幅として性能評価モデル (ex. ルーフラインモデル) でも頻繁に利用される

```
! STREAM Triad (stream.f)
PARAMETER (n=100000000,offset=0,ndim=n+offset,ntimes=10)
DOUBLE PRECISION a(ndim),b(ndim),c(ndim) ! static array
...
!$OMP PARALLEL DO
    DO 60 j = 1,n
        a(j) = b(j) + scalar*c(j) ! Triad
60    CONTINUE
```

メモ

コードのバイトと浮動小数点演算数との比(いわゆるB/F)を評価 → A64FXのハードウェアスペックとしてのB/Fと比較することを推奨

STREAM: first touch policyを満足するような配列の初期化が適用 (確認すること)

STREAMによるメモリバンド幅の測定 (01_bandwidth) (1/5) [C]



- STREAM: メモリからのデータ転送速度 (実効的なメモリバンド幅)を計測するベンチマーク
(<https://www.cs.virginia.edu/stream>)

- ◆ 4個のカーネル (Copy, Scale, Add, Triad)から構成

- 本講習会ではTriadに着目

- 浮動小数点演算の数: 2 per loop
 - メモリロード (=の右辺): 2*8bytes per loop,
メモリストア (=の左辺): 1*8bytes per loop

- ◆ 連続的なメモリアクセスのパターンを有するカーネルを計測

- 実効的なメモリバンド幅として性能評価モデル (ex. ルーフラインモデル) でも頻繁に利用される

```
/* STREAM Triad (stream.c) */
static STREAM_TYPE      a[STREAM_ARRAY_SIZE+OFFSET],
                      b[STREAM_ARRAY_SIZE+OFFSET],
                      c[STREAM_ARRAY_SIZE+OFFSET]; /* static array */
...
#pragma omp parallel for
    for (j=0; j<STREAM_ARRAY_SIZE; j++)
        a[j] = b[j]+scalar*c[j]; /* Triad */
#endif
```

メモ

コードのバイトと浮動小数点演算数との比(いわゆるB/F)を評価 → A64FXのハードウェアスペックとしてのB/Fと比較することを推奨

STREAM: first touch policyを満足するような配列の初期化が適用 (確認すること)



STREAMによるメモリバンド幅の測定 (01_bandwidth) (2/5)

■ 設定

- ◆ データサイズ: 2288 MiB (配列サイズ = 100000000) > L2全体(=32MiB)
 - 配列1個あたりのサイズを100000000に設定
 - CMG内 (8GiB) に収まる程度に設定
- ◆ 時間計測: 各カーネルを10回繰り返し測定 (NTIMESはデフォルトから変更なし)
- ◆ Thread affinityの設定
 - OMP_PLACES=cores
 - (スレッド数) <= 12: OMP_PROC_BIND=close → CMG内にスレッドを収める
 - (スレッド数) > 12 : OMP_PROC_BIND=spread → ノード内にスレッドができるだけ均等に配置する
- ◆ ページングポリシーの設定
 - XOS_MMM_L_PAGING_POLICY=demand:demand:prepage
 - STREAM: 静的配列でメモリを確保 → 1番目の値が意味をもつ (demandに設定)

■ ポイント

- ◆ スレッド数の増大による性能 (メモリバンド幅, GB/s) の変化を観察する
- ◆ メモリ律速なコードの性能向上に寄与するコンパイルオプション (高速ストア: -Kzfill, -ffj-zfill) の効果を確認する

STREAMによるメモリバンド幅の測定 (01_bandwidth) (3/5)



■ ログの出力例 (omp*)

- ◆ マルチスレッドの利用でバンド幅の計測値が上昇
(単位時間あたり大量のデータをメモリから移動できる)

```
# omp1 (1スレッド) [Fortran (frtpx -Kfast -Kopenmp -Kzfill)]
# XOS_MMM_L_PAGING_POLICY=demand:demand:prepage
Number of Threads = 1
. . .
Function      Rate (MB/s)  Avg time   Min time   Max time
Copy:          24775.85    0.0646     0.0646     0.0646
Scale:         20955.28    0.0764     0.0764     0.0764
Add:           35280.72    0.0680     0.0680     0.0681
Triad:          32684.16    0.0734     0.0734     0.0735
```

Triadの計測値

メモリバンド幅 (平均)経過時間 (sec.)

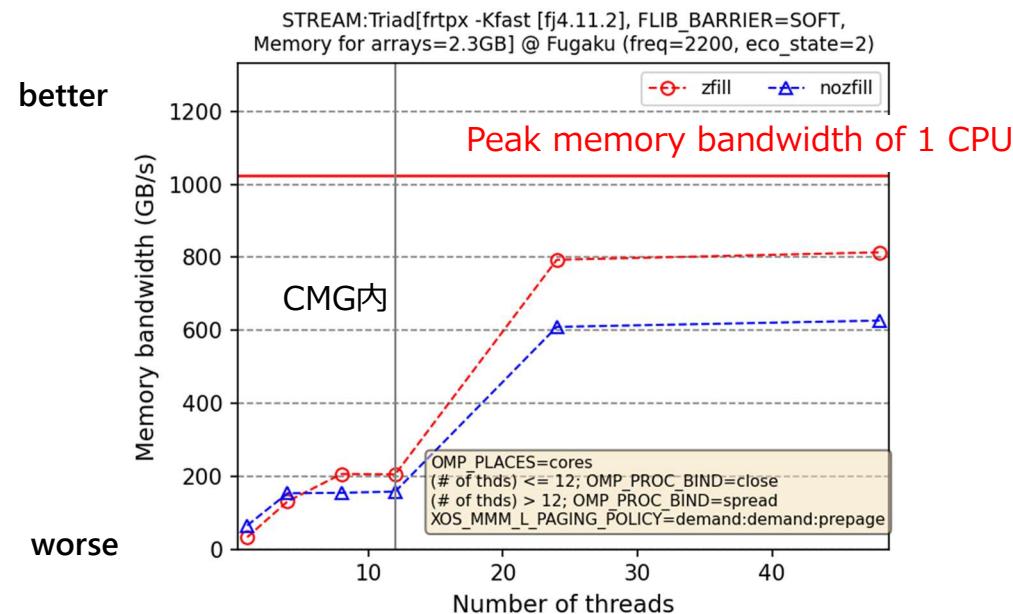
```
# omp48 (48スレッド) [Fortran (frtpx -Kfast -Kopenmp -Kzfill)]
# XOS_MMM_L_PAGING_POLICY=demand:demand:prepage
Number of Threads = 48
. . .
Function      Rate (MB/s)  Avg time   Min time   Max time
Copy:          825548.83   0.0019     0.0019     0.0020
Scale:         811768.04   0.0020     0.0020     0.0020
Add:           822211.03   0.0029     0.0029     0.0029
Triad:          816608.23   0.0029     0.0029     0.0030
```

Triadの計測値

STREAMによるメモリバンド幅の測定 (01_bandwidth) (4/5)

■ 結果 [Fortran]

- ◆ スレッドの増加に従いバンド幅が増加
- ◆ 高速ストアのOFF (**nozfill**) に対しON (**zfill**) で性能向上を確認 (>12thd: 1.3倍の速度向上)
 - 高速ストアON (**zfill**)の48スレッドで800GB/s (~80% of peak performance)を確認



STREAMによるメモリバンド幅の測定 (01_bandwidth) (5/5)

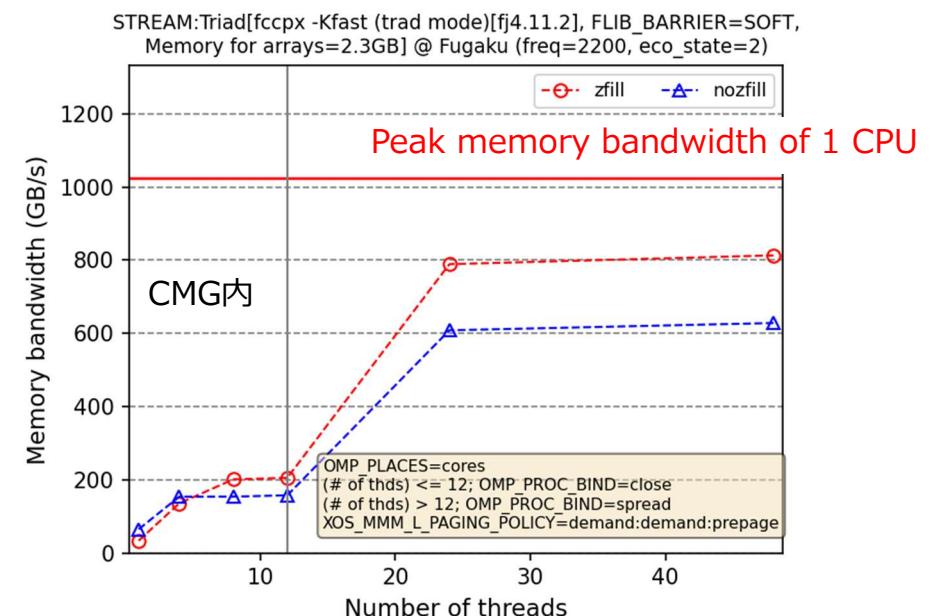
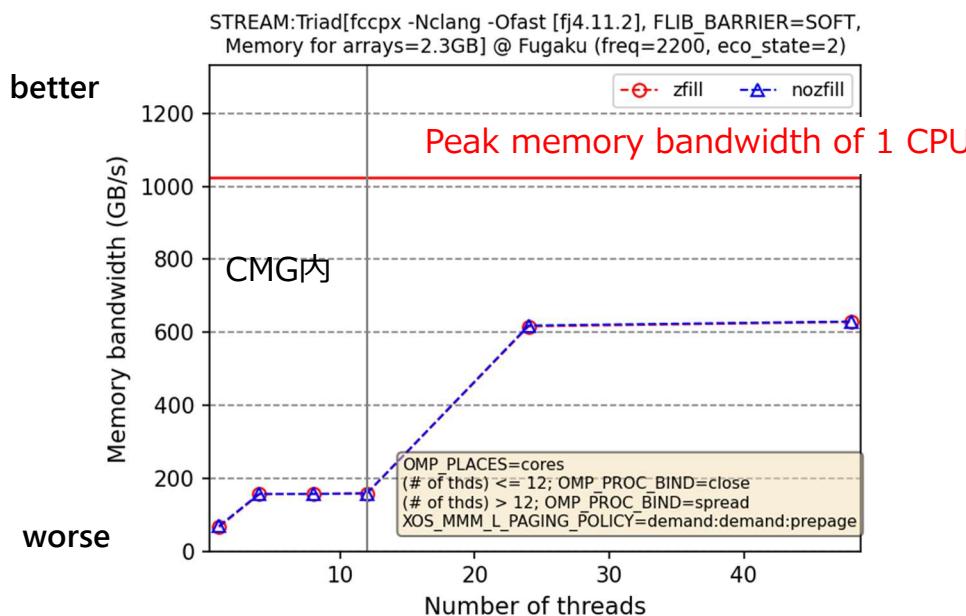


■ 結果 [C]

- ◆スレッドの増加に従いバンド幅が増加
- ◆Tradモード(右): 高速ストアのOFF ([nozfill](#)) に対し
ON ([zfill](#)) で性能向上を確認 (>12thd: 1.3倍の速度向上)
 - 高速ストアON ([zfill](#))の48スレッドで~800GB/s (~80% of peak performance)を確認
- ◆Clangモード(左): Triadでは高速ストアの効果はほぼ見られない (cf. Addでは効果を確認)

メモ

Clangモードにおける高速ストアの効果 in Add (48スレッド)
626795.12 MB/s (zfillなし) → 817403.95 MB/s(zfillあり)



Thread affinity (02_thread-affinity) (1/2)



■ 設定

- ◆ 01_bandwidthと同じ (Thread affinityを除く)
 - 高速ストア (zfill) を有効にして測定
- ◆ Thread affinityの設定
 - すべてのスレッド数に対し OMP_PROC_BIND=close → スレッド0のまわりに配置される
 - 12スレッド以上でも特定のCMGの周辺に配置されている可能性あり
 - 12スレッド以下ならCMG内にスレッドが収まる
 - すべてのスレッド数に対し OMP_PROC_BIND=spread → ノード内にできるだけ均等に配置される
 - 12スレッド以下でもCMG内に収まっているとは限らない

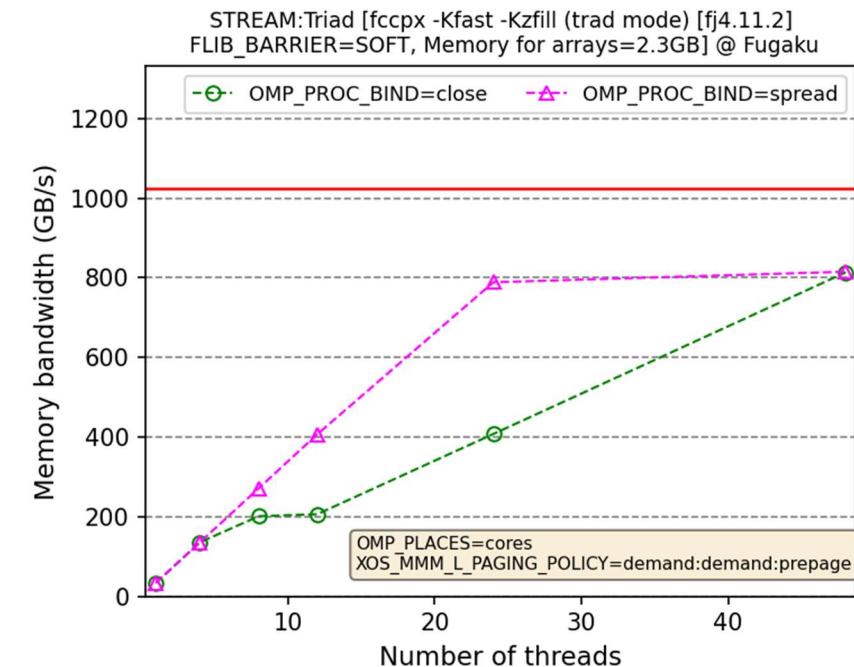
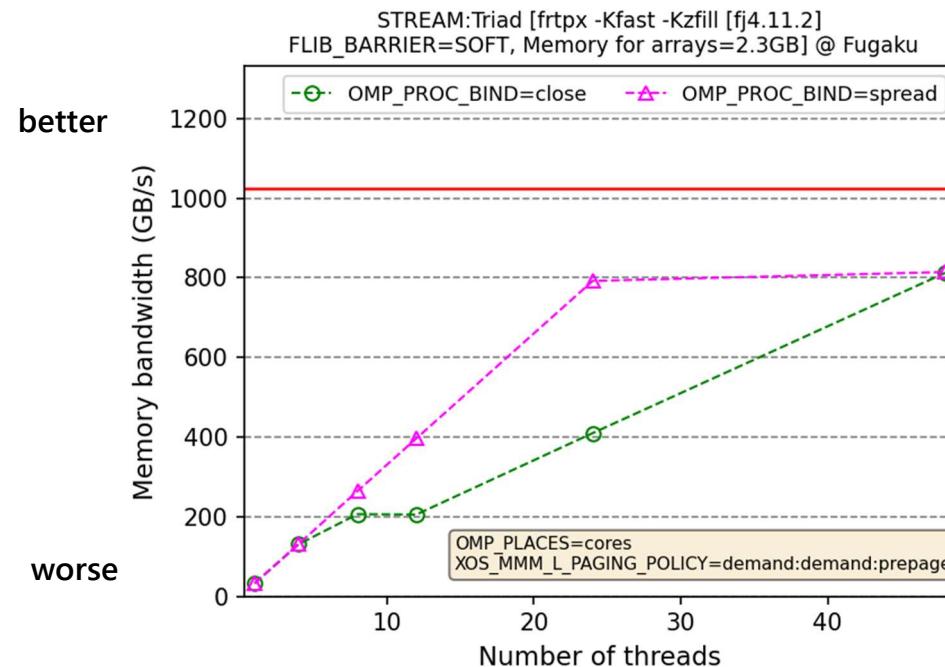
■ ポイント

- ◆ コアバインディングの設定の変更によりバンド幅の測定値が受ける影響を観察する

Thread affinity (02_thread-affinity) (2/2)

■ 結果 [Fortran (左図), C Trad mode (右図)]

- ◆ **OMP_PROC_BIND=close**: CMG以上のスレッド数 (>12)で **spread**からの性能劣化が顕著
 - 48スレッドで全ての物理コアを使用する状況では問題なし
- ◆ **OMP_PROC_BIND=spread**: **close**よりメモリアクセスの性能は良好
- ◆ **close**と**spread**のどちらがよいか? → 並列計算の設定に依存
 - 例: MPI+OpenMP (<=12スレッド) の場合は**close**が好ましいと考えられる



メモ

- C Clangモードの結果: 解釈が難しい状況
- C Tradでの計測を推奨
 - 実験結果 → Appendixを参照



ハンズオンの内容 (1/2)

■ 03_shared-memory/02_thread-affinityに取り組んでください。

- ◆ まず、Readme.mdに目を通してください。
- ◆ 課題の実行には、STREAMのコンパイルが必要です。
03_shared-memory/00_streamに行って、まずSTREAMのコンパイルを行ってください。

■ Exercise A:

- ◆ E1: fj_zfill.close と fj_zfill.spreadのtask.shの違いを確認する。
- ◆ E2: STREAM (stream.exe) を実行し、Triadのバンド幅の値が、スレッド数に応じてそれぞれどう変化するか確認する。

■ Exercise Aが終わったら

- ◆ Exercise Bに取り組んでみてください。
- ◆ あるいはほかの課題に取り組んでみてください (次ページ)

ハンズオンの内容 (2/2)



■ 他の課題

◆ 03_shared-memory/01_bandwidth

- STREAMベンチマークを使ったメモリバンド幅の測定

◆ 03_shared-memory/03_data-locality (Appendix)

- STREAMベンチマークを使った、「富岳」のpaging policy の影響の確認

◆ A64FXのNUMA構成の確認

- 計算ノードに入って numactl -H もしくは lscpu

◆ EPCC OpenMP Microbenchmark Suiteを使ったOpenMPのオーバーヘッドの測定

- <https://github.com/EPCCed/epcc-openmp-microbenchmarks>

◆ 02_cache/02_mm-vs-mv

- 行列行列積の、DGEMMとほかの実装による性能の比較 (Appendix)

- -SSL2の代わりに-SSL2BLAMPを使用して、スレッド並列を使用したDGEMMのscalabilityの測定。

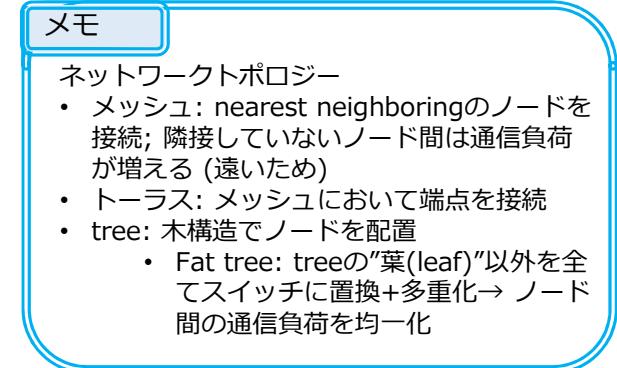
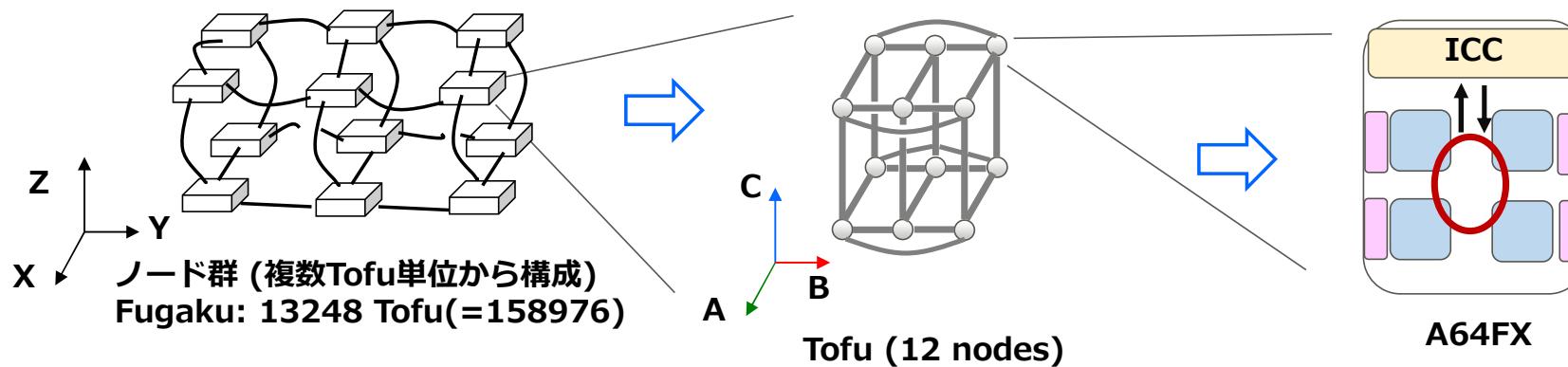
Theme 4: Pure-MPI execution

対応する実習

- [04_pure-mpi/](#)

システムの概要: インターコネクト (1/4)

- ノード群: 複数のTofu単位から構成 (Tofu 1個: X,Y,Z-axesで座標指定)
 - ◆ 隣接Tofu間が互いに“接続”するようネットワークが構成
- 1 Tofu (12ノードから構成; 2x3x2 in A,B,C-axes) がノード群の基本ユニット
 - ◆ Tofu内はA軸, C軸がメッシュ, B軸がトーラス (端と端は互いに接続)
 - A=0, 1; B=0, 1, 2; C=0,1
→ Tofu内において任意のノードは4本の線で互いに接続
- 1ノード (CPU)からInterconnect Controller (ICC)を通して他のノードと接続
 - ◆ ノードの物理座標: 6D (X, Y, Z, A, B, C)で表現



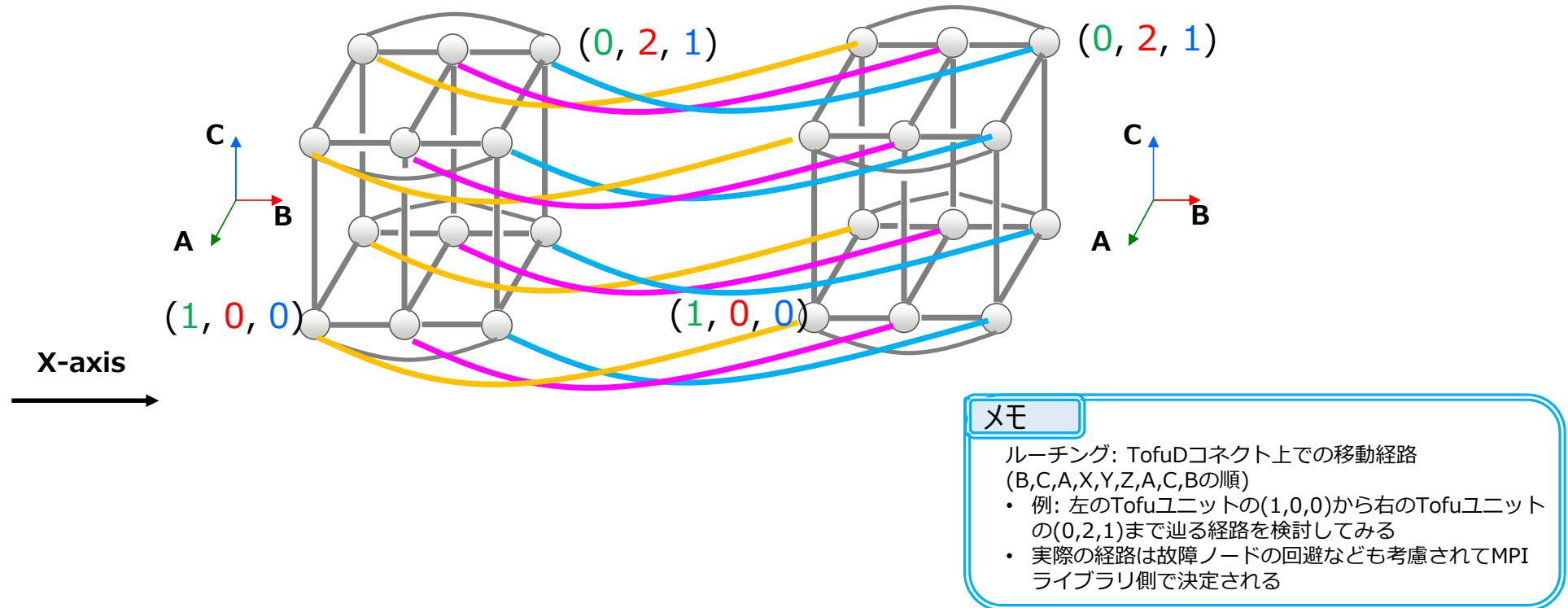


システムの概要: インターコネクト (2/4)

■ 隣接Tofu間の接続

- ◆ X, Y, Z-axesの各々について、隣接Tofu間で同一Tofu座標のノードを接続させる

例: X-axisの場合で隣接Tofu間接続を図示

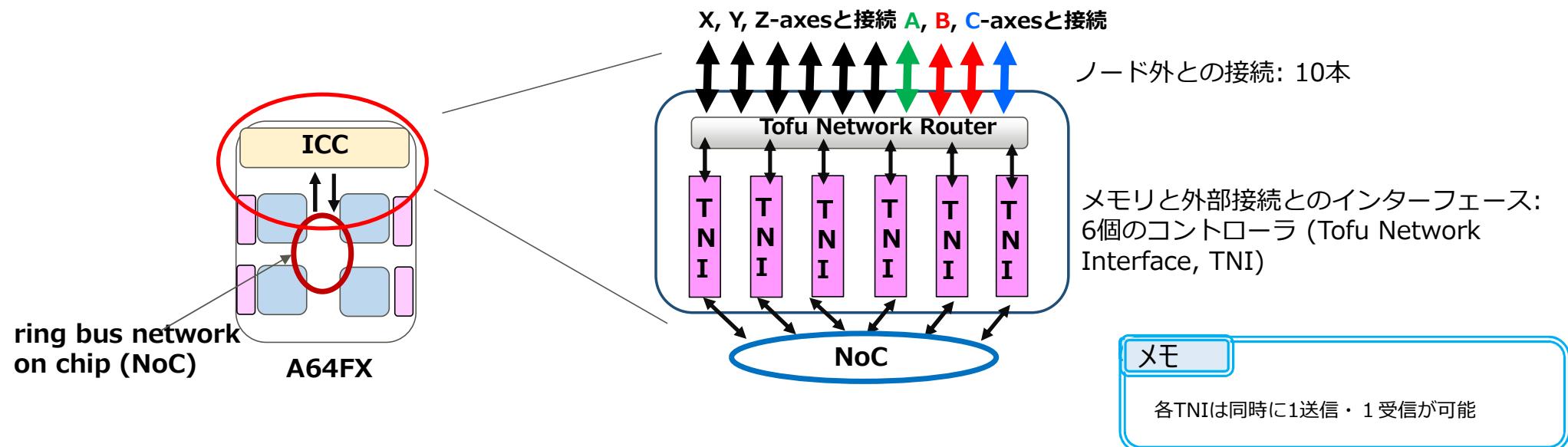




システムの概要: インターコネクト (3/4)

■ CPU側: Interconnect controller (ICC)の内部

- ◆ Tofu Network Interface (TNI): 6個搭載 → メモリと外部への送受信をつなぐハードウェア
- ◆ 外部への送受信: 10本のライン
 - Tofu内の座標用: 4本 (A, B+, B-, C)
 - 隣接Tofu間の座標用: 6本 (X+, X-, Y+, Y-, Z+, Z-)



システムの概要: インターコネクト (4/4)



■ 「富岳」利用におけるインターコネクトに対する考え方

◆ TNI: MPIのプロセス通信を担う重要なハードウェア

- 装置の存在と物量(6個)を認識すれば十分
→ コーディングやプログラム実行で陽に制御しなくてよい
 - ハードウェアとしての機能をうまく使うことができる条件を把握しておくのは重要

メモ

TNIの積極的な利用例 (advanced)
• p2p通信における複数TNI利用, uTofu

◆ ノードの割当: 6Dの物理座標を制御する必要はない → 3D (X, Y, Z) で対応可能

- Tofuを意識したノード割当 → “12”の倍数のノード数と形状指定で対応
 - 3D+トーラスモードでのノード割当例: 2x2x12:torus (12の倍数; 48ノードでTofu単位が隣接するよう割当)
 - cf. 1D (形状指定なし): 48 → ノード割当は状況依存
- 実際のノード割当ポリシー: 「富岳」ウェブサイトを確認
 - 例: (X, Y, Z)は回転することもある: 2x2x12, 12x2x2, 2x12x2 も可能かもしれない。
- 実用上: ジョブの入りやすさも重要
 - strictな形状指定が求められる例: 大規模実行において通信がボトルネックとなる場合



Theme 4における着眼点

■ ノード間の通信に着目

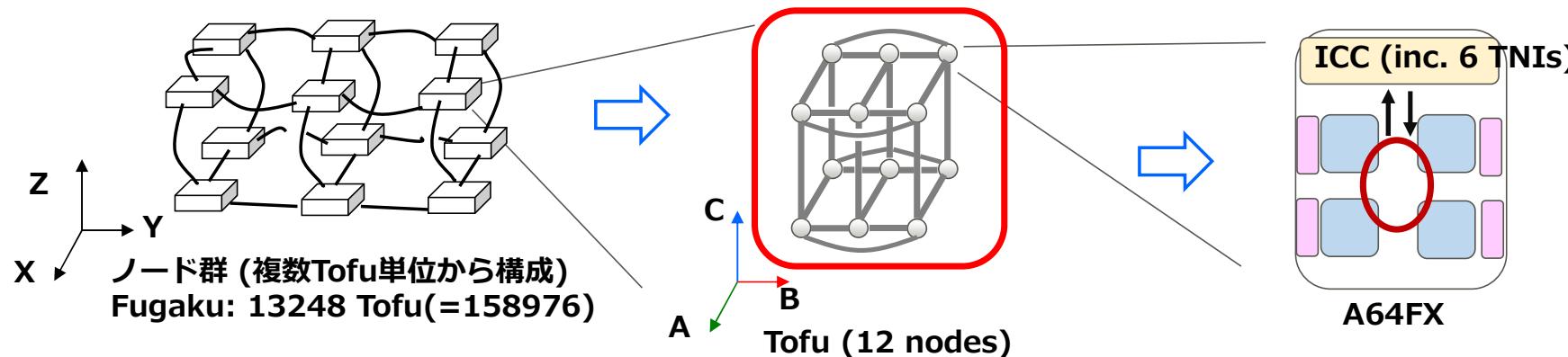
- ◆ ノード間の通信が発生する状況でプログラムを動かす
→ インターコネクトの特性を調べるパターン

■ 理解のポイント

- ◆ max-proc-per-node (or PPN, Processes per node)
 - ノードあたりのMPIプロセス数
- ◆ Tofuバリア通信機能
 - 集団通信ルーチンの実行で利用可能なTofuインターフェクトに特有の機能
(ハードウェアサポートのバリア)

メモ

- ノード間通信のレイテンシ: μs のオーダー
- 01_pingpongおよび02_mpingpongの実習で計測可能
(appendix)



集団通信とバリア通信機能 (03_tofu-barrier) (1/7)



■ MPIの集団通信ルーチン

- ◆(コミュニケーションに属する) 全てのプロセスが一斉に通信 → 完了するまで待ち合わせ(同期)

■ Tofuバリア通信機能

- ◆ハードウェア(TNIに設けられたバリアゲート)で同期処理
→ 対応する集団通信ルーチンの高速化につながる可能性
 - MPI_Barrier, MPI_Bcast, MPI_Reduce, MPI_Allreduceで利用可能

◆ 使用の確認方法

- MPI統計情報ファイルの“Tofu Barrier Collective Communication Count”で確認可能
 - プログラム実行時で“OMPI_MCA_mpi_print_stats>=1”により統計情報ファイルが出力

◆ 使用条件あり (詳細はマニュアルを要確認)

- 大まかな条件
 - TNIのバリアゲートが確保できる (物量が足りないと使用されない)
 - コミュニケータはintra-communicator
(inter-communicatorに関係するものは不可)
 - コミュニケータに関係するプロセス数やノード数が一定数以上
(一定数は“4”が基本; 詳細な条件あり)
 - データのバイト (配列の要素数) に制約 (MPI_Barrier以外)
→ “短い”メッセージ長が基本

メモ

intra-communicatorの例

- MPI_COMM_WORLD
- MPI_COMM_WORLDからMPI_Comm_splitで分割されて生成された各コミュニケーション

inter-communicatorの例

- MPI_Intercomm_createで生成されたコミュニケーション

集団通信とバリア通信機能 (03_tofu-barrier) (2/7)



■ カーネル: 3種類を検討 (MPI_Barrier, MPI_Bcast, MPI_Allreduceを含む)

◆ elm3

- MPI_Allreduce+MPI_SUMで倍精度配列3要素を送受信

◆ elm6

- MPI_Allreduce+MPI_SUMで倍精度配列6要素を送受信

◆ elm6.sep

- elm6について, MPI_Allreduceを2回に分けて (i.e., 倍精度配列3要素 × 2) 送受信

◆ MPI_Reduce, MPI_Allreduceのバリア通信機能を使うための詳細な条件

- リダクション演算で「演算順序を保証しない」必要がある

➢ 環境変数OMPI_MCA_coll_base_reduce_commute_safeで変更可能 (1:保証する, 0:保証しない(デフォルト))

- メッセージの要素数の制限を守る必要がある

➢ 許容される要素数は演算の種類やデータ型に依存する

➢ MPI_SUMで倍精度(8バイト)の場合 → 3要素がバリア通信機能利用の上限

メモ

MCAパラメータcoll_tbi_repeat_maxの設定により要素数の上限の変更は可能



集団通信とバリア通信機能 (03_tofu-barrier) (3/7) [Fortran]

```
! Code fraction (elm3/main.F90)
do it = 1, 500000
  ! main kernel
  if ( me .eq. 0 ) then ! rank=0

    u_init = 0.0D0 + 1.0D-6*it
    call MPI_BCAST(u_init, 1,
MPI_DOUBLE, MPI_ROOT, workercomm, ierr)

    else                      ! other ranks

      call MPI_BCAST(u_init, 1,
MPI_DOUBLE, 0, workercomm, ierr)
      ...
      call MPI_ALLREDUCE(buff, A, 3,
MPI_DOUBLE, MPI_SUM, localcomm, ierr)

    end if

    call MPI_BARRIER(MPI_COMM_WORLD,ierr)
end do
```

```
! On Communicators (common b/w elm3, elm6, and elm6.sep)
mygroup = 1
if (me .eq. 0 ) then ! rank=0
  mygroup = 0
end if
call MPI_COMM_SPLIT(MPI_COMM_WORLD, mygroup, 1, localcomm, ierr)
...
call MPI_INTERCOMM_CREATE(localcomm, 0, MPI_COMM_WORLD, &
& mypartner, inter_group, workercomm, ierr);
```

```
! Code fraction (elm6/main.F90)
call MPI_ALLREDUCE(buff, A, 6,
MPI_DOUBLE, MPI_SUM, localcomm, ierr)
```

```
! Code fraction (elm6.sep/main.F90)
call MPI_ALLREDUCE(buff(1), A(1),
3, MPI_DOUBLE, MPI_SUM, localcomm, ierr)
call MPI_ALLREDUCE(buff(4), A(4),
3, MPI_DOUBLE, MPI_SUM, localcomm, ierr)
```

備考(重要): 一度に送らずに細かく分けて通信
→ 通常のMPI通信関数の利用では"やってはいけない"こと

集団通信とバリア通信機能 (03_tofu-barrier) (3/7) [C]



```
/* Code fraction (elm3/main.c) */
for ( int it = 0; it <=500000 ; ++it ) {
    /* main kernel */
    if ( me == 0 ) { /* rank=0 */

        u_init = 0.0 + 1.0e-6*it;
        MPI_Bcast(&u_init, 1, MPI_DOUBLE,
MPI_ROOT, workercomm);

    } else {           /* other ranks */

        MPI_Bcast(&u_init, 1, MPI_DOUBLE,
0, workercomm);
        ...
        MPI_Allreduce(buff, A, 3,
MPI_DOUBLE, MPI_SUM, localcomm);
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

```
/* On Communicators (common b/w elm3, elm6, and elm6.sep) */
int mygroup = 1;
if ( me == 0 ) mygroup = 0; /* rank = 0 */
MPI_Comm_split(MPI_COMM_WORLD, mygroup, 1, &localcomm);
...
MPI_Intercomm_create(localcomm, 0, MPI_COMM_WORLD,
mypartner, inter_group, &workercomm);
```

```
/* Code fraction (elm6/main.c) */
    MPI_Allreduce(buff, A, 6,
MPI_DOUBLE, MPI_SUM, localcomm);
```

```
/* Code fraction (elm6.sep/main.c) */
    MPI_Allreduce(&buff[0], &A[0], 3,
MPI_DOUBLE, MPI_SUM, localcomm);
    MPI_Allreduce(&buff[3], &A[3], 3,
MPI_DOUBLE, MPI_SUM, localcomm);
```

備考(重要): 一度に送らずに細かく分けて通信
→ 通常のMPI通信関数の利用では"やってはいけない"こと



集団通信とバリア通信機能(03_tofu-barrier) (4/7)

■ 設定

- ◆ PPN=4 (#PJM --mpi "max-proc-per-node=4")
- ◆ ノード数: 12ノード (1 Tofuに相当)
- ◆ メッセージ長
 - MPI_Bcast: 倍精度で1要素 (バリア通信機能の条件を満足)
 - MPI_Allreduce: 倍精度で3要素 (Tofuバリア通信機能の条件を満足), or 倍精度で6要素 (条件を満足しない)
- ◆ 時間計測: 各カーネルを繰り返し実行し, プロセスごとに経過時間を測定
(fortran: 500000回, C: 500001回)
 - カーネル実行の末端でMPI_Barrierを呼び出し → 全てのプロセスで(ほぼ)同じ経過時間になることが期待
- ◆ MPI統計情報ファイルを出力
 - プログラム実行時の設定: "OMPI_MCA_mpi_print_stats=2, OMPI_MCA_mpi_print_stats_ranks=0,1"
 - ランク0とランク1のみMPI統計情報を出力させる

■ ポイント

- ◆ プログラム中の集団通信ルーチンがバリア通信機能を使用可能か条件を検討する
- ◆ MPI統計情報ファイルの内容に基づき結果を分析する

集団通信とバリア通信機能(03_tofu-barrier) (5/7)



■ ログの出力例 (log.*)

```
# rank=0からの出力 (log.0000) in elm3
mygroup= 0
World: rank      = 0 size      = 48
Split: rank(local)= 0 size(local)= 1
Inter: size(remote)= 47
u_init = 0.500000000000000
A( 0) = 0.000000
A( 1) = 0.000000
A( 2) = 0.000000
Elapsed time of main part (s) = 11.628
```

```
# rank=1からの出力 (log.0001) in elm3
mygroup= 1
World: rank      = 1 size      = 48
Split: rank(local)= 0 size(local)= 47
Inter: size(remote)= 1
u_init = 0.500000000000000
A( 1) = 23.500000
A( 2) = 1151.500000
A( 3) = 2279.500000
Elapsed time of main part (s) = 11.627
```

経過時間 (sec.)

注意: MPI_Barrierを呼び出しているため,
各ランクはほぼ同じタイミングで処理を終える

集団通信とバリア通信機能(03_tofu-barrier) (6/7)

■ MPI統計情報の例 (err.*.)

メモ

- 注意: mpi_print_stats=1の場合 → rank=0が関わった集団通信の情報のみが出力
- 今回のカーネル: MPI_Comm_splitでコミュニケーションを(two disjoint setsに)分離 ($\{0\} + \{1, 2, \dots\}$) → ランクごとにMPI統計情報を出力する必要があった

rank=0のMPI統計情報 (err.*.0)

```
----- MPI Information -----
Dimension           1
Shape              12

----- MPI Memory Usage (MiB) -----
Estimated_Memory_Size   89.06
...
----- Tofu Barrier Collective Communication Count --
Bcast                0
Reduce               0
Allreduce             0
```

rank=1のMPI統計情報 (err.*.1)

```
----- MPI Information -----
Dimension           1
Shape              12

----- MPI Memory Usage (MiB) -----
Estimated_Memory_Size   91.23
...
----- Tofu Barrier Collective Communication Count --
Bcast                0
Reduce               0
Allreduce             500001
```

rank=0ではバリア通信機能が使用されていない (MPI_Barrierを除く)

Bcast, Reduce, Allreduceでバリア通信機能が使用された回数

検討: Bcastでバリア通信機能が使われない理由は?

集団通信とバリア通信機能(03_tofu-barrier) (7/7)



■ 結果 [Fortran] (Cも同傾向)

- ◆ elm6に比べelm6.sep (Allreduceを2回に分けてコール) で高速化を確認
 - 30%程度の負荷削減
 - バリア通信機能の有効利用が性能にimpactがあったことを示唆
 - Allreduceを2回呼び出す → カウントが $2^*(\text{カーネル繰り返し回数})$ に変化している
- ◆ (集団) 通信がプログラムのボトルネックになっているときには有効な手法と期待
 - プログラムにおいてバリア通信機能の有効性を確認する簡便方法
 - “export MCA_OMPI_coll=^tbi”をプログラム実行時にセット
 - 全てのバリア通信機能がOFF → 性能の劣化の程度を観察

	経過時間 (sec) (rank=0)	Tofu Barrier Collective Communication count (rank=1)		
		Bcast	Reduce	Allreduce
elm3	11.656	0	0	500000
elm6	19.438	0	0	0
elm6.sep	13.275	0	0	1000000

(※ブーストエコモード (freq=2200, eco_state=2) で測定)



ハンズオンの内容 (1/2)

- 04_pure-mpi/03_tofu-barrierに取り組んでください。
 - ◆ まず、Readme.mdに目を通してください。

- exercise A:
 - ◆ E1: elm3/main.cに目を通し、どのMPI集団通信ルーチンが使われているか確認する。
 - ◆ E2: elm6/main.cとelm6.sep/main.c に関しても、同様に確認する。
 - ◆ E3: MPI統計情報を確認して、tofuバリア通信機能が使われているかどうかを確認する。
 - ◆ E4: elm3, elm6, elm6.sepの性能比較をする。

- Exercise Aが終わったら
 - ◆ Exercise Bに取り組んでみてください。
 - ◆ あるいはほかの課題に取り組んでみてください (次ページ)



ハンズオンの内容 (2/2)

■ 他の課題

- ◆ 04_pure-mpi/00_imb
 - IMB (intel MPI benchmark) のインストール
(<https://github.com/intel/mpi-benchmarks.git>)
- ◆ 04_pure-mpi/01_pingpong (see Appendix)
 - IMBを使ったPingPongによる2ノード間測定
 - Eager通信 vs Rendezvous通信
- ◆ 04_pure-mpi/02_mpinglepong (see Appendix)
 - 1ノード当たり複数プロセスを使ったIMBの PingPong の測定
 - 1ノード当たりのプロセス数を増やした時のバンド幅の変化の観測
- ◆ 別の実装 (04_pure-mpi/simple-p2p)によるPingPongの測定
- ◆ IMBで集団通信ルーチン (Bcast, Gather, Alltoall)に関する性能調査。
 - 注意：ノード数を増やすときは、メモリー不足に陥ることを避けるため、PPN=4としてください。
- ◆ A64FX向けチューニング技術検討会で発表された資料に目を通す。
 - https://www.hpci-office.jp/en/events/seminars/meeting_A64FX_220127
 - MPI, LLI0に関する話題

Theme 5: MPI+OpenMP hybrid parallel calculations

対応する実習

- [05_mpi-openmp/](#)

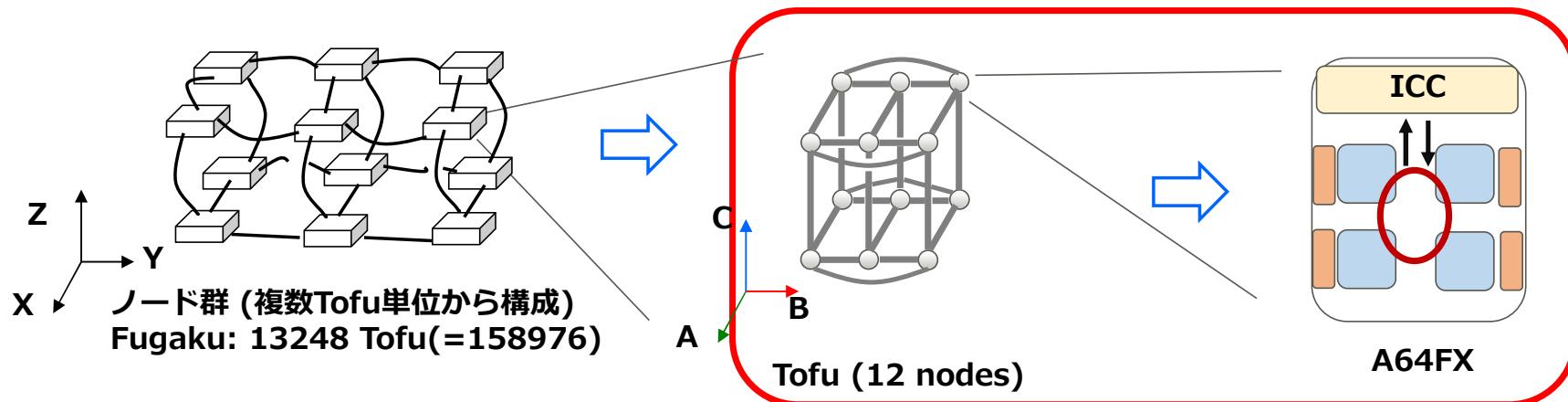
Theme 5の着眼点



- ノード内の共有メモリ並列計算とノード間の通信に着目
 - ◆ スレッド並列を利用しつつ、ノード間の通信が発生する状況でプログラムを動かす
→ 「富岳」で柔軟に並列計算(MPI+OpenMP)を実行する方法を調べるパターン

■ キーワード

- ◆ MPI_Init_thread
 - OpenMPの並列領域からMPI通信関数を呼び出す場合に使用する初期化用関数
 - MPIライブラリごとに許容される"thread level"を認識したコーディングが必要
- ◆ OpenMP並列領域からのMPI通信関数呼び出し
 - "独立"な処理のオーバラップ





富士通MPIにおけるスレッドのレベル (02_thread-level) (1/2) [Fortran]

■ カーネル: 1種類

- ◆ MPI_Init_threadの呼び出しで要求するスレッドのレベルを指定
→ 要求レベルが許容されているかを調べる
 - 要求レベル: required, MPI側で許容されるレベル: provided
 - required > provided → 要求されたレベルは許容できない

```
! Code fraction (main.F90)
call MPI_INIT_THREAD (required, provided, ierr)

if ( required .gt. provided ) then
  ...
  call MPI_FINALIZE()
  stop
end if
```

メモ

スレッド並列領域からMPI通信関数を呼び出すレベル
(MPI規格)

- MPI_THREAD_SINGLE: MPI通信関数の呼び出しは許可しない (MPI_Initと同じ)
- MPI_THREAD_FUNNELED: プライマリースレッドからのMPI通信関数の呼び出しを許可
- MPI_THREAD_SERIALIZED: 複数のスレッドからのMPI通信関数の呼び出しを許可. ただし逐次的であることを要請
- MPI_THREAD_MULTIPLE: 複数のスレッドからのMPI通信関数の呼び出しを許可

Intel MPI Benchmark (IMB)のIMB-MT

- MPI_THREAD_MULTIPLEの利用が前提

- MPI forum: <https://mpi-forum.org/>

富士通MPIにおけるスレッドのレベル (02_thread-level) (1/2) [C]

■ カーネル: 1種類

- ◆ MPI_Init_threadの呼び出しで要求するスレッドのレベルを指定
→ 要求レベルが許容されているかを調べる
 - 要求レベル: required, MPI側で許容されるレベル: provided
 - required > provided → 要求されたレベルは許容できない

```
/* Code fraction (main.c) */
MPI_Init_thread (NULL, NULL, required, &provided);

if ( required > provided ) {
    ...
    MPI_Finalize()
    return EXIT_FAILURE;
}
```

メモ

スレッド並列領域からMPI通信関数を呼び出すレベル
(MPI規格)

- MPI_THREAD_SINGLE: MPI通信関数の呼び出しは許可しない (MPI_Initと同じ)
- MPI_THREAD_FUNNELED: プライマリースレッドからのMPI通信関数の呼び出しを許可
- MPI_THREAD_SERIALIZED: 複数のスレッドからのMPI通信関数の呼び出しを許可. ただし逐次的であることを要請
- MPI_THREAD_MULTIPLE: 複数のスレッドからのMPI通信関数の呼び出しを許可

Intel MPI Benchmark (IMB)のIMB-MT

- MPI_THREAD_MULTIPLEの利用が前提

- MPI forum: <https://mpi-forum.org/>



富士通MPIにおけるスレッドのレベル (02_thread-level) (2/2)

■ 結果 (out.*)

- ◆ 富士通MPIではMPI_THREAD_MULTIPLEはサポートしていないことを確認

- MPI_THREAD_SERIALIZED → 逐次的であれば、複数のスレッドからMPI通信関数を呼び出すことは許可
 - 例: 特定のスレッドIDのみからMPI通信関数を呼び出す → OK
 - 例: omp criticalで指定されたブロックでMPI通信関数を呼び出す → OK

```
# 要求レベルがMPI_THREAD_SERIALIZEDの場合
# out-2.*.0
Required thread level is acceptable for
your MPI.
Required level:    2
Provided level:   2
Each integer indicates:
* 0: MPI_THREAD_SINGLE
* 1: MPI_THREAD_FUNNELED
* 2: MPI_THREAD_SERIALIZED
* 3: MPI_THREAD_MULTIPLE
```

```
# 要求レベルがMPI_THREAD_MULTIPLEの場合
# out-3.*.0
Error: Required thread level is higher
than the supported one in your MPI
Required level:    3
Provided level:   2
Each integer indicates:
* 0: MPI_THREAD_SINGLE
* 1: MPI_THREAD_FUNNELED
* 2: MPI_THREAD_SERIALIZED
* 3: MPI_THREAD_MULTIPLE
```

- HPCプログラミングセミナー https://www.hpci-office.jp/events/seminars/seminar_text

スレッド並列領域中のMPI通信 (03_hybrid) (1/4)

■ カーネル: 構造格子の差分法 (2次元のステンシル計算) を模したパターン (前進微分のみ評価)

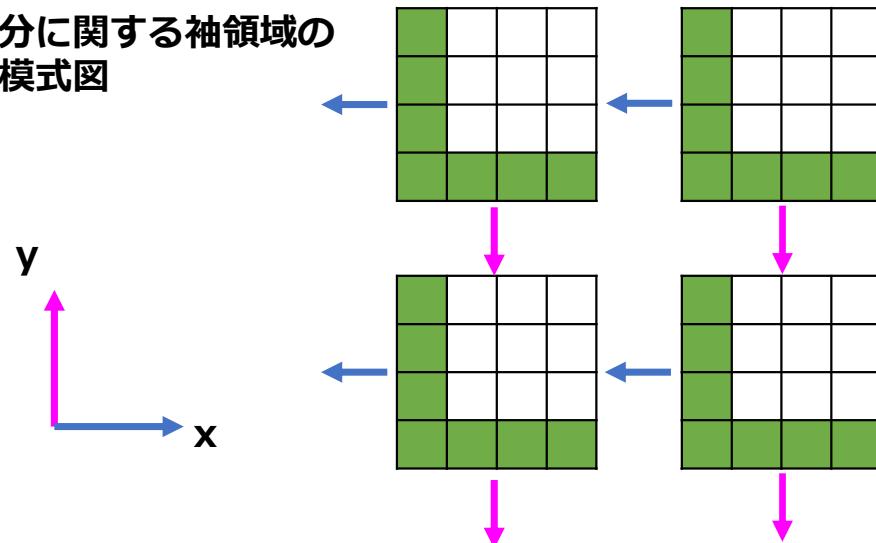
◆ sendrecv

- 袖領域 (Halo) の通信にMPI_Sendrecvを使用; スレッド並列は未使用(pure-mpi)

◆ thread_nt2

- sendrecvと同じ袖領域通信をスレッド並列領域で呼び出し; 2スレッドで固定
 - スレッド0: 通信と演算, スレッド1: (通信データと関係ない) 演算 → オーバラップ

前進微分に関する袖領域の送信の模式図



メモ

実際には4種類のカーネルが用意 (計算しているものは全て同じ)

- nonblocking: MPI_Isend, MPI_Irecvを使ったパターン
- sendrecv: MPI_Sendrecvを使ったパターン
- serial: 逐次コード
- thread_nt2: スレッド並列領域内でMPI_Sendrecvを呼び出すパターン; スレッド数は2に固定

$$\Delta_{x+}f(x, y) = f(x + \Delta x, y) - f(x, y)$$

$$\Delta_{y+}f(x, y) = f(x, y + \Delta y) - f(x, y)$$

$$\text{境界条件 (B.C.) : } f(\partial\Omega) = 0$$

スレッド並列領域中のMPI通信 (03_hybrid) (2/4) [Fortran]



```
! Code fraction (sendrecv/main.F90)
! p2p communication along x-axis
call MPI_SENDRECV ( send_buff, nysize_local,
MPI_DOUBLE, rank_dest_x, tag, recv_buff,
nysize_local, MPI_DOUBLE, rank_src_x, tag,
cartComm, stat, ierr) ! communication
call forward_dx_mpi (nxend1, nxsta, nxend,...,
recv_buff, f, fdx)    ! calculations
...
! p2p communication along v-axis
call MPI_SENDRECV (send_buff, nxsize_local,
MPI_DOUBLE, rank_dest_y, tag, recv_buff,
nxsize_local, MPI_DOUBLE, rank_src_y, tag,
cartComm, stat, ierr) ! communication
call forward_dy_mpi (nyend1, nxsta, nxend,...,
recv_buff, f, fdy)    ! calculations
! calculations, independent of recv_buff, f,
fdx, and fdy
call source_term_mpi (nxsta, nxend, nysta,
nyend, dx, dy, it, s)
```

```
! Code fraction (thread_nt2/main.F90)
!$OMP PARALLEL NUM_THREADS(2) &
!$OMP & PRIVATE(ix,iy,ii,tid)
tid = omp_get_thread_num()
...
if ( tid .eq. 0 ) then
  ! thread 0
  ! p2p communication along x-axis
  ...
! p2p communication along y-axis
else
  ! thread 1
  ...
end if
!$OMP END PARALLEL
```

スレッド並列領域中のMPI通信 (03_hybrid) (2/4) [C]



```
/* Code fraction (sendrecv/main.c) */
/* p2p communication along x-axis */

    MPI_Sendrecv (&send_buff[0], max_buffer_size,
    MPI_DOUBLE, rank_dest_x, tag, &recv_buff[0],
    max_buffer_size, MPI_DOUBLE, rank_src_x , tag,
    cartComm, &status);           /* communication */
    forward_dx_mpi (nxsize_local1,nxsize_local,
..., &recv_buff[0], f, fdx); /* calculations */

...
/* p2p communication along y-axis */

    MPI_Sendrecv (&send_buff[0], max_buffer_size,
    MPI_DOUBLE, rank_dest_y, tag, &recv_buff[0],
    max_buffer_size, MPI_DOUBLE, rank_src_y , tag,
    cartComm, &status);           /* communication */
    forward_dy_mpi (nysize_local1,nxsize_local,
..., &recv_buff[0], f, fdy); /* calculations */

/* calculations, independent of recv_buff, f,
fdx, and fdy */

    source_term_mpi (nxsize_local, nysize_local,
offset_x, offset_y, dx, dy, it, s);
```

```
/* Code fraction (thread_nt2/main.c) */
#pragma omp parallel num_threads(2)
{
    int tid = omp_get_thread_num();
...
    if ( tid == 0 ) {
        /* thread 0 */
        /* p2p communication along x-axis */
        /* p2p communication along y-axis */
    } else {
        /* thread 1 */
    }
}
```



スレッド並列領域中のMPI通信 (03_hybrid) (3/4)

■ 設定

◆ 並列設定 (sendrecv)

- PPN=4 (#PJM --mpi "max-proc-per-node=4")
- ノード数: 12ノード

◆ 並列設定 (thread_nt2)

- PPN=4 (#PJM --mpi "max-proc-per-node=4")
- ノード数: 12ノード
- スレッド数: 2

◆ データサイズ: 10000 × 10000

◆ 時間計測: 反復回数 (NITR) を10000, プロセス0の経過時間を測定

■ ポイント

◆ 通信パターンの変更による性能の変化を観察する

スレッド並列領域中のMPI通信 (03_hybrid) (4/4)



■ 結果 (out.*.0の出力をまとめ)

◆ PPN=4の場合: sendrecvに比べthread_nt2 で性能向上 (~33-35%程度の負荷低減)

- スレッド並列の併用 (処理のオーバラップ) で一定の効果を確認

◆ 批判的な観察

- 使用コア数を揃えた比較 (96コア) → sendrecvのほうがthread_nt2より速い

➤ sendrecv: 48MPIから96MPIで速度向上は~2倍 → MPI並列で良くスケールを示唆

- 処理のオーバラップの適用: オーバラップする処理の負荷を評価し, 検討する必要がある

➤ コーディングのパターンとして覚えておくのがよい

	Node数	PPN	スレッド数	Elapsed time of main loop (s)	
				Fortran	C
sendrecv	12	4	1	37.3	37.6
thread_nt2	12	4	2	24.3	25.2
sendrecv	12	8	1	19.3	20.5

※ブーストエコモードでの測定 (freq=2200, eco_state=2)

ハンズオンの内容 (1/2)



■ 05_mpi-opemp/03_hybridに取り組んでください。

- ◆ まずReadme.mdに目を通してください。

■ Exercise A:

- ◆ thread_nt2/main.cで使用するスレッドレベルは、富士通MPIで許容されるものか確認する。
- ◆ thread_nt2/main.cとsendrecv/main.cのMPI通信に関する箇所を比較する。

■ Exercise Aが終わったら

- ◆ Exercise Bに取り組んでみてください。
- ◆あるいはほかの課題に取り組んでみてください（次ページ）

ハンズオンの内容 (2/2)



■ 他の例：

- ◆ 05_mpi-openmp/02_thread-level

- 富士通MPIで許容されるスレッドレベルのチェック

- ◆ 05_mpi-openmp/01_vcoord ([Appendix](#))

- デフォルトのランク割り当て方法の確認
 - vcoordfileを利用したランク割り当て制御

- ◆ vcoordfileを使って, MPMD(Multi Program Multi Data) モデルのプログラムを実行するジョブスクリプトを作成する

- ◆ 「富岳」に整備されているプログラムのscalabilityを調査してみる。

https://www.hpci-office.jp/for_users/appli_info

Theme 6: LLIO

対応する実習

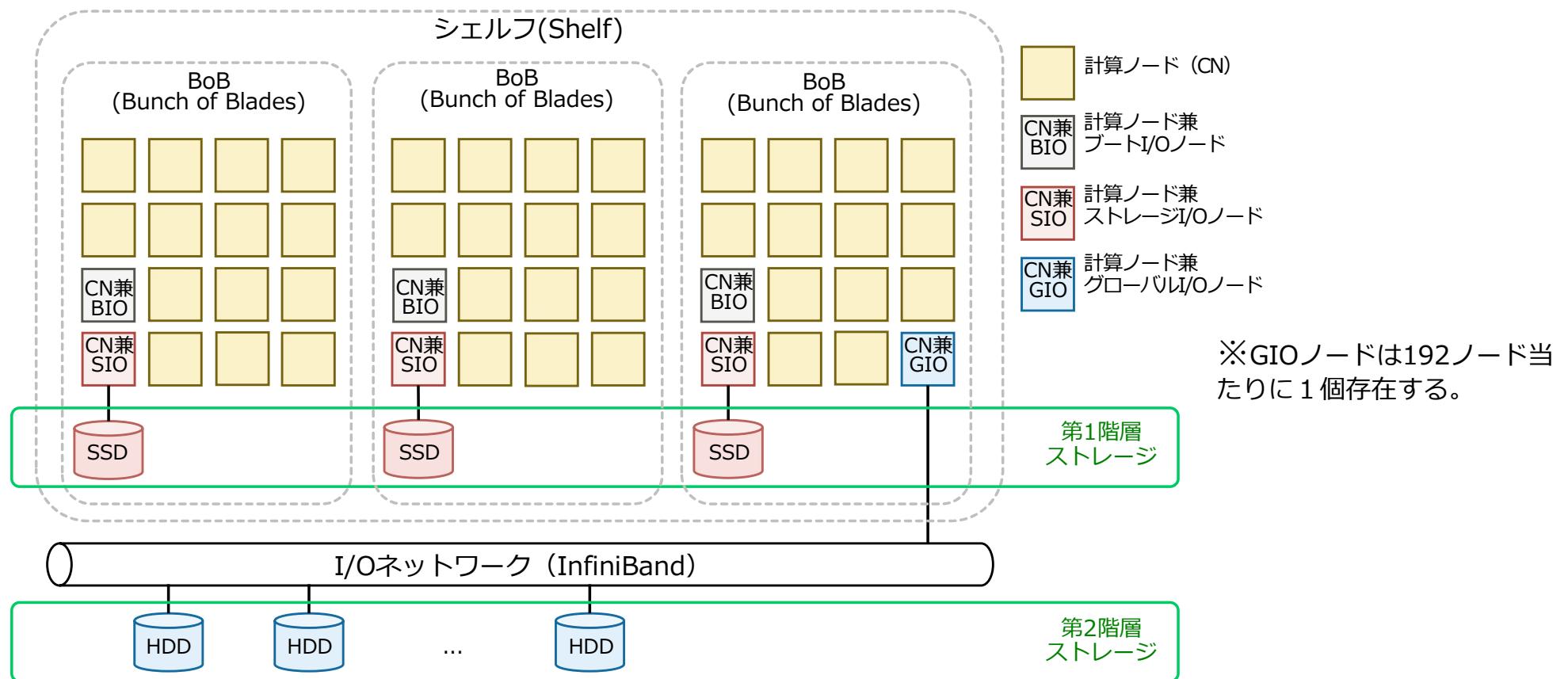
- [01_localtmp/](#)
- [02_sharedtmp/](#)

「富岳」のストレージ構成 (1/3)



■ 「富岳」のストレージ構成は下図の通り

- ◆ 第1階層ストレージと第2階層ストレージはSIO (ストレージI/O) ノードと、GIO (グローバルI/O) ノードを介して行われる



「富岳」のストレージ構成 (2/3)



■ ノードにはそれぞれノードIDが割り振られている

- ◆ ジョブ統計情報ファイルには、使用したノードのノードIDが16進数で出力される

```
NODE ID (USE) : 0x2D1A000D 0x2D1A0010 0x2D020008 0x2D240010
```

- ◆ ノードがBIO, SIOまたはGIOノードであるかはノードIDから確認することが可能

- BIO: ノードIDの末尾が 01
- SIO: ノードIDの末尾が 02
- GIO: 以下のURLに示されている

https://www.fugaku.r-ccs.riken.jp/doc_root/ja/contents/0603-2020/GIO_LIST.txt

- ◆ idcheckコマンドにより、ノードがBIO, SIOまたはGIOであるかを確認することが可能

- -nオプション: ノードIDをコンマで区切って指定
- -fオプション: ノードIDが記載されたファイルを指定

```
[_LNlogin]$ idcheck -n 0x2D1A000D,0x2D1A0001,0x2D1A0002  
0x2D1A0001 BIO  
0x2D1A0002 SIO  
0x2D1A000D CN
```

「富岳」のストレージ構成 (3/3)

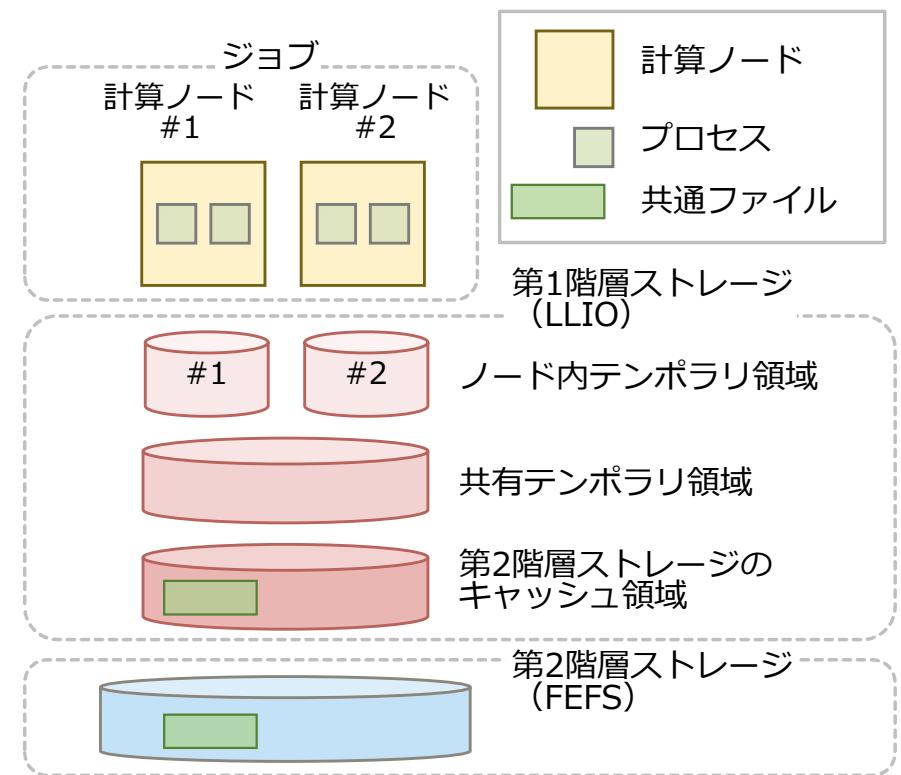


■ 第1階層ストレージ

- ◆ LLIO (Lightweight Layered IO-Accelerator) によって管理されるSSDストレージ
- ◆ ジョブ開始時に初期化され、ジョブ終了時に削除される
- ◆ 以下の3つの領域を持つ：
 - 第2階層ストレージのキャッシュ領域
 - ノード内テンポラリ領域
 - 共有テンポラリ領域

■ 第2階層ストレージ

- ◆ Lusterの技術を基にしたFEFS (Fujitsu Exascale Filesystem)により管理されるHDDストレージ
- ◆ ログインノードおよび各計算ノードで共有される
- ◆ 大容量のデータや長期保存するデータに適している



「富岳」のストレージ構成 (2/2)

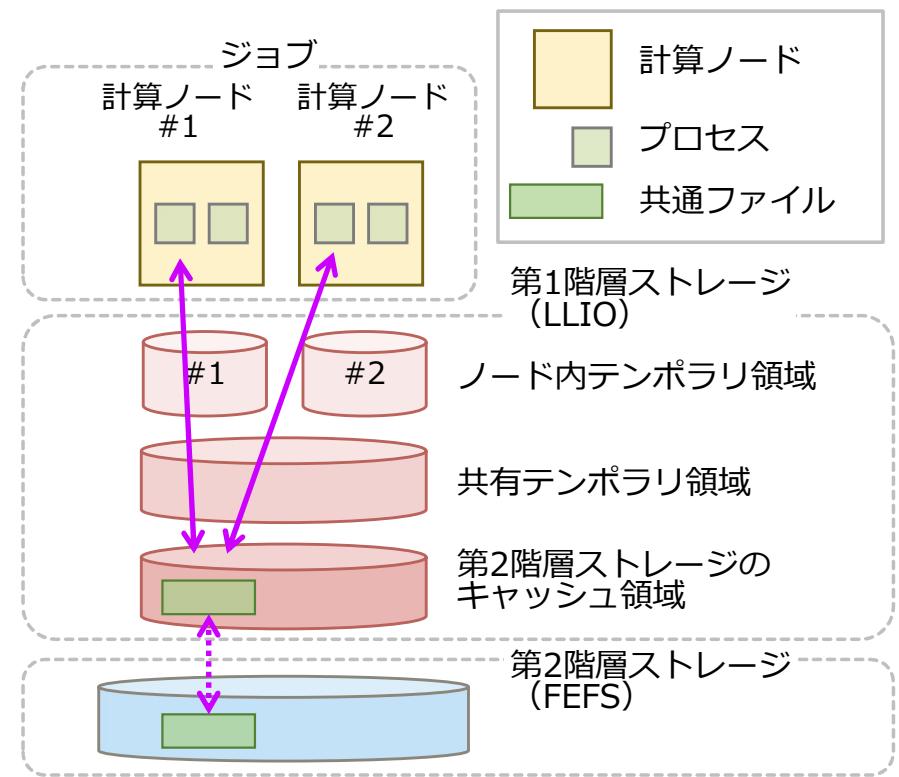


■ 第1階層ストレージ

- ◆ LLIO (Lightweight Layered IO-Accelerator) によって管理されるSSDストレージ
- ◆ ジョブ開始時に初期化され、ジョブ終了時に削除される
- ◆ 以下の3つの領域を持つ：
 - 第2階層ストレージのキャッシュ領域
 - ノード内テンポラリ領域
 - 共有テンポラリ領域

■ 第2階層ストレージ

- ◆ Lusterの技術を基にしたFEFS (Fujitsu Exascale Filesystem)により管理されるHDDストレージ
- ◆ ログインノードおよび各計算ノードで共有される
- ◆ 大容量のデータや長期保存するデータに適している

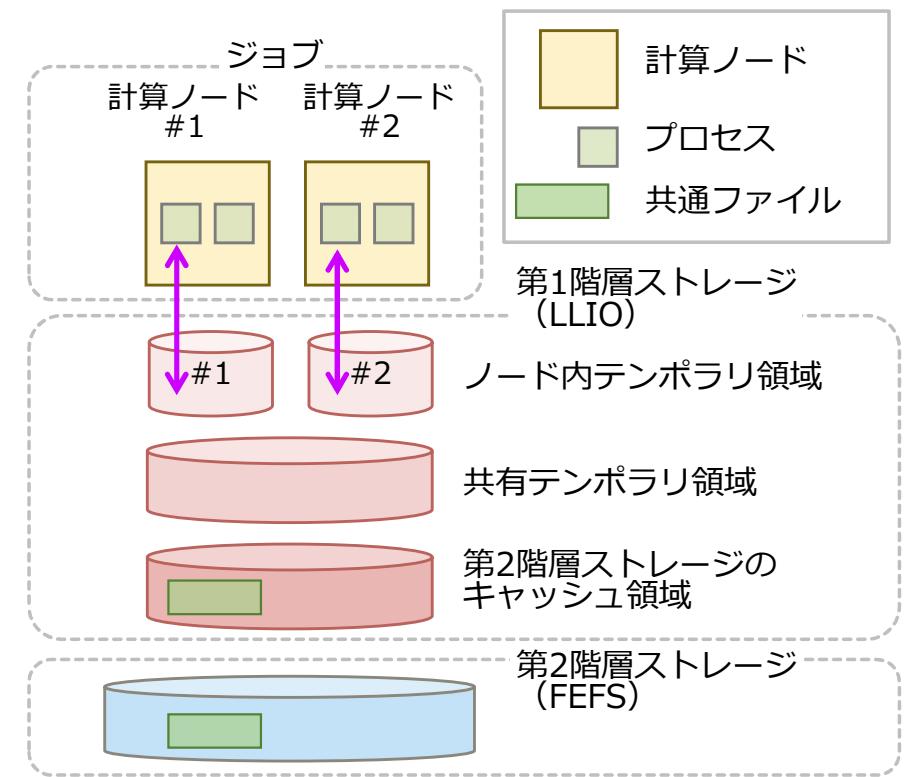


「富岳」のストレージ構成 (2/2)



■ 第1階層ストレージ

- ◆ LLIO (Lightweight Layered IO-Accelerator) によって管理されるSSDストレージ
- ◆ ジョブ開始時に初期化され、ジョブ終了時に削除される
- ◆ 以下の3つの領域を持つ：
 - 第2階層ストレージのキャッシュ領域
 - ノード内テンポラリ領域
 - 共有テンポラリ領域



■ 第2階層ストレージ

- ◆ Lusterの技術を基にしたFEFS (Fujitsu Exascale Filesystem)により管理されるHDDストレージ
- ◆ ログインノードおよび各計算ノードで共有される
- ◆ 大容量のデータや長期保存するデータに適している

「富岳」のストレージ構成 (2/2)

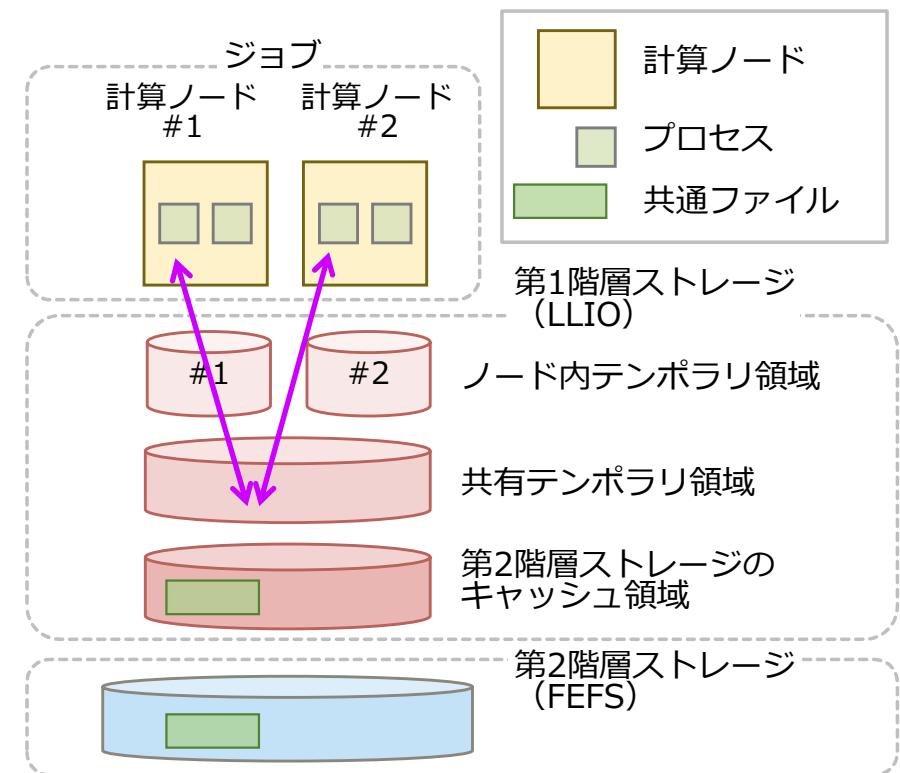


■ 第1階層ストレージ

- ◆ LLIO (Lightweight Layered IO-Accelerator) によって管理されるSSDストレージ
- ◆ ジョブ開始時に初期化され、ジョブ終了時に削除される
- ◆ 以下の3つの領域を持つ：
 - 第2階層ストレージのキャッシュ領域
 - ノード内テンポラリ領域
 - 共有テンポラリ領域

■ 第2階層ストレージ

- ◆ Lusterの技術を基にしたFEFS (Fujitsu Exascale Filesystem)により管理されるHDDストレージ
- ◆ ログインノードおよび各計算ノードで共有される
- ◆ 大容量のデータや長期保存するデータに適している。





第1階層ストレージの3つの領域

- 第1階層ストレージは、ジョブに割り当てられた計算ノードからの参照範囲などが異なる3種類の領域を持つ。

◆ LLIOを利用してことで、I/Oの負荷を低減できる可能性がある。

- ◆ 多数のノードでプログラムを実行する場合
- ◆ 大規模なファイルを扱う場合

◆ 利用するプログラムの特性に応じて、これらの領域を使い分けると良い。

メモ

多数ノードでプログラムを実行する場合は、実行ファイルや入力ファイル等をlio_transferコマンド（共通ファイル配布機能）を利用して、SIOノードへのアクセス負荷を分散させること

領域名	参照範囲	適用事例	最大容量	容量の指定法
ノード内テンポラリ領域	同一ジョブの同一ノード内	ランク・ノードごとに独立した中間ファイルや一時ファイルの作成・参照	全領域で 87GiB /node	pjsubのオプションにて指定 ※指定しない場合0MiB
共有テンポラリ領域	同一ジョブの全ての計算ノード	ランク・ノード間でのファイルを参照、巨大なファイルの操作		pjsubのオプションにて指定 ※指定しない場合0MiB
第2階層ストレージのキャッシュ領域	同一ジョブの全ての計算ノード	標準出力・標準エラー出力など、第2階層に保存されるファイルの出力		87GiB - (ノード内テンポラリ領域 + 共有テンポラリ領域) ※128MiB以上の容量が必要 ※ジョブのキャッシュ利用量に対して小さい場合、キャッシュミス頻発の可能性あり

- Performance of File System; https://www.fugaku.r-ccs.riken.jp/doc_root/ja/user_guides/perf/Perf-FS.pdf



■ ファイル入出力についての検討事項

- ◆ 不必要な出力はないか
- ◆ ファイル出力頻度は適切か（ログ出力、リスタートファイルの出力などの頻度）
- ◆ ノード内テンポラリ領域や共有テンポラリ領域の利用

■ ファイル出力のコストを見積もる方法

- ◆ ファイル出力の頻度を変えてみる
- ◆ 時間計測ルーチンを使用して、ファイル出力に関する箇所の処理時間を計測する
- ◆ ファイルI/Oに関するプロファイリングツールを使用する
 - 「富岳」では、スケーラブルなHPC向けI/O評価ツール darshanが提供されている
 - darshan: アルゴンヌ大学で開発されている Open-Source Software
<https://www.mcs.anl.gov/research/projects/darshan/>



■ ファイルの出力に焦点

◆複数プロセスからファイルを出力するパターン

- ノード内テンポラリ領域 (localtmp) を利用するジョブでは、各プロセスが個別に出力するプログラムで調査
- 共有テンポラリ領域 (sharedtmp) を利用するジョブでは、MPI-IOを利用して各プロセスからのデータをマージして出力するプログラムで調査

■ 理解のポイント

◆LLIOのファイル書き出し処理への効果

プログラムのI/O解析 (1/5) [Fortran]



■サンプルプログラムの概要

- ◆ 実行ファイルの引数に入力ファイル名を指定

- ◆ 入力ファイルでは、

- 配列 A のサイズ (array_size)
- 計算ループの回転数 (n_loop)
- ファイル出力の頻度 (i_skip)
- 出力ファイル名 (output_file)

を指定する。

- ◆ ループ内で計算を実行し、その結果を i_skip ステップごとに、各プロセスがバイナリファイルに出力する

- ◆ 01_localtmp での出力ファイル名は、指定したファイル名に各プロセス番号が付与されたものである (output_0001.txt 等)。

```
call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierr)

if (command_argument_count() >= 1) then
    call get_command_argument(1, argv)
    input_file = adjustl(argv)
    open (unit=input_no, file=input_file, ...)

    read(input_no,*) array_size
    read(input_no,*) n_loop
    read(input_no,*) i_skip
    read(input_no,'(A)') output_file
    close(unit=input_no)
else
    ...
endif
...
call add_myrank_to_filename(myrank, output_file)
```

プログラムのI/O解析 (1/5) [C++]



■サンプルプログラムの概要

- ◆ 実行ファイルの引数に入力ファイル名を指定

- ◆ 入力ファイルでは、

- 配列 A のサイズ (array_size)
- 計算ループの回転数 (n_loop)
- ファイル出力の頻度 (i_skip)
- 出力ファイル名 (output_file)

を指定する。

- ◆ ループ内で計算を実行し、その結果を i_skip ステップごとに、各プロセスがバイナリファイルに出力する

- ◆ 01_localtmp での出力ファイル名は、指定したファイル名に各プロセス番号が付与されたものである (output.txt_0 等)。

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (argc > 1) {
    input_file = argv[1];
    std::ifstream infil(input_file, std::ios::in);

    getline(infil, tmp); array_size = stoi(tmp);
    getline(infil, tmp); n_loop = stoi(tmp);
    getline(infil, tmp); i_skip = stoi(tmp);
    getline(infil, tmp); output_file = tmp;

    infil.close();
} else {
    . .
}
. .
output_file = add_myrank_to_filename(myrank, output_file);
```

プログラムのI/O解析 (2/5) [Fortran]



■サンプルプログラムの概要(01_localtmp/fortran/main.f90) (続き)

```
real(kind=8), allocatable :: A(:)
. . .
allocate(A(1:array_size))
. . .
MPI_Barrier(MPI_COMM_WORLD, ierr)
total1 = MPI_Wtime()
do ii = 1, n_loop
    if (mod(ii, i_skip) == 0) then
        call MPI_Barrier(MPI_COMM_WORLD, ierr)
        t1 = MPI_Wtime()
        open (unit=output_no, file=output_file, status='replace', &
              ,action='write', form='unformatted', access='stream')
        write(output_no) A
        close(unit=output_no)
        call MPI_Barrier(MPI_COMM_WORLD, ierr)
        t2 = MPI_Wtime()
        if (myrank == main_rank) &
            write(6,'(4x,"II=",i6,2x, "ELAPSED(SEC)=“,1PE15.6)') ii, t2-t1
    endif
enddo
call MPI_Barrier(MPI_COMM_WORLD, ierr)
total2 = MPI_Wtime()
if (myrank == main_rank) &
    write(6,'("NPROCS=“,i6,2x,”TOTAL ELAPSED(SEC)=“,1PE15.6)’) nprocs, total2 - total1:
```

ファイル出力にかかった
時間を計測

プログラムのI/O解析 (2/5) [C++]



■サンプルプログラムの概要(01_localtmp/cpp/main.cpp) (続き)

```
double *A;  
. . .  
A = new double[array_size];  
. . .  
MPI_Barrier(MPI_COMM_WORLD);  
total1 = MPI_Wtime();  
for (ii = 1; ii <= n_loop; ii++) {  
    if (ii&i_skip == 0) {  
        MPI_Barrier(MPI_COMM_WORLD);  
        t1 = MPI_Wtime();  
        ofs = fopen(output_file.c_str(), "wb");  
        fwrite(A, sizeof(double), array_size, ofs);  
        fclose(ofs);  
  
        MPI_Barrier(MPI_COMM_WORLD);  
        t2 = MPI_Wtime();  
        if (myrank == main_rank) {  
            std::cout << "    II= " << ii << "    ELAPSED(SEC) = " << t2 - t1 << std::endl;  
        }  
    }  
}  
MPI_Barrier(MPI_COMM_WORLD);  
total2 = MPI_Wtime();  
if (myrank == main_rank) {  
    std::cout << " NPROCS = " << nprocs << "    TOTAL ELAPSED(SEC) = " << total2 - total1 << std::endl;  
}
```

ファイル出力にかかった
時間を計測

プログラムのI/O解析 (3/5) [Fortran]



■サンプルプログラムの概要(02_sharedtmp/fortran/main.f90) (続き)

```
real(kind=8), allocatable :: A(:)
. . .
allocate(A(1:array_size))
. . .
call MPI_Barrier(MPI_COMM_WORLD, ierr)
total1 = MPI_Wtime()
do ii = 1, n_loop
    if (mod(ii, i_skip) == 0) then
        call MPI_Barrier(MPI_COMM_WORLD, ierr)
        t1 = MPI_Wtime()
        offset = (array_size * 8.d0) * myrank
        call MPI_File_open(MPI_COMM_WORLD, output_file, ...
        call MPI_File_write_at(ifh, offset, A, array_size, MPI_REAL8, ...
        call MPI_File_close(ifh, ierr)
        call MPI_Barrier(MPI_COMM_WORLD, ierr)
        t2 = MPI_Wtime()
        if (myrank == main_rank) &
            write(6,'(4x,"II=",i6,2x, "ELAPSED(SEC)=",1PE15.6)') ii, t2-t1
    endif
enddo
call MPI_Barrier(MPI_COMM_WORLD, ierr)
total2 = MPI_Wtime()
if (myrank == main_rank) &
    write(6,'("NPROCS=",i6,2x,"TOTAL ELAPSED(SEC)=",1PE15.6)') nprocs, total2 - total1
```

ファイル出力にかかった
時間を計測

プログラムのI/O解析 (3/5) [C++]



■サンプルプログラムの概要(02_sharedtmp/cpp/main.cpp) (続き)

```
double *A;
. . .
A = new double[array_size];
. . .
MPI_Barrier(MPI_COMM_WORLD);
total1 = MPI_Wtime();
for (int ii = 1; ii <= n_loop; ii++){
    if (ii&i_skip == 0) {
        MPI_Barrier(MPI_COMM_WORLD);
        t1 = MPI_Wtime();
        offset = (array_size * 8.0) * myrank;
        MPI_File_open(MPI_COMM_WORLD, output_file.c_str(), ...
        MPI_File_write_at(ifh, offset, A, array_size, MPI_DOUBLE, ...
        MPI_File_close(&ifh)
        MPI_Barrier(MPI_COMM_WORLD)
        t2 = MPI_Wtime()
        if (myrank == main_rank) {
            std::cout << "    II= " << ii << "  ELAPSED (SEC)= " << t2-t1 << std::endl;
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
    total2 = MPI_Wtime();
    if (myrank == main_rank) {
        std::cout << " NPROCS = " << nprocs << "  LOOP ELAPSED (SEC)= " << total2 - total1 << std::endl;
    }
}
```

ファイル出力にかかった
時間を計測



プログラムのI/O解析 (4/5)

- darshan: スケーラブルなHPC向けI/O評価ツール
- 「富岳」におけるdarshanの使用方法
 - ◆ I/Oプロファイリングデータを取得するジョブスクリプト例

```
#!/bin/sh
. . .
#PJM --mpi proc=4
#PJM -x PJM_LLIO_GFSCACHE=/vol0004
#PJM -S
. . .

. /vol0004/apps/oss/spack/share/spack/setup-env.sh
spack load darshan-runtime%fj scheduler=fj                                darshanはspackにて提供

export DARSHAN_LOG_DIR_PATH=./darshan_output
# export DARSHAN_ENABLE_NON_MPI=1    ### non-MPIの場合は追加          プロファイリングデータの出力先を指定
. . .

mpicexec -np ${PJM_MPI_PROC} -std-proc out -x LD_PRELOAD=libdarshan.so ${bindir}/run.x input.txt
```



プログラムのI/O解析 (5/5)

■ 「富岳」におけるdarshanの使用方法（続き）

- ◆ darshan-runtimeによって取得されたI/Oプロファイリングデータはログインノードから、 darshan-parser コマンドで確認できる。

```
[_LNlogin] . /vol0004/apps/oss/spack-v0.21/share/spack/setup-env.sh
[_LNlogin] spack load darshan-util@3.4.0 arch=linux-rhel8-cascadelake
[_LNlogin] darshan-parser --file-list darshan_output/usr_run.x_JOBID.darshan
. . .
# Per-file summary of I/O activity.
# -----
# <record_id>: darshan record id for this file
# <file_name>: full file name
# <nprocs>: number of processes that opened the file
# <slowest>: (estimated) time in seconds consumed in IO by slowest process
# <avg>: average time in seconds consumed in IO (including metadata) per process

# <record_id> <file_name> <nprocs> <slowest> <avg>
0001112223334445556  /vol0004/.../input.txt 1 0.000657 0.000657
1112223334445556667  /local/output_0000.txt 1 26.508906 26.508906
2223334445556667778  /local/output_0001.txt 1 28.176589 28.176589
. . .
```

ノード内テンポラリ領域の利用(1/2)



■ ノード内テンポラリ領域を使用するジョブスクリプト例

```
#!/bin/sh
. . .
#PJM --llio localtmp-size=1Gi # 必要なノード内テンポラリ領域のサイズを指定
#PJM --mpi "max-proc-per-node=4"
#PJM -S
. . .
cat << EOD > ./input.txt
2621440 # 配列A のサイズ => ファイルサイズ2621440 Byte /1024/1024 * 8 = 20 MiB
100      # n_loop
5        # i_skip
${PJM_LOCALTMP}/output.txt # ジョブ中で ${PJM_LOCALTMP}は /local に展開される。
EOD
mpiexec -np ${PJM_MPI_PROC} -std-proc ./results/%J.log ${bindir}/run.x input.txt

# mpiexec -np ${PJM_MPI_PROC} -std-proc log sh -c \
#   'if [ ${PLE_RANK_ON_NODE} -eq 0 ]; then \
#     mv ${PJM_LOCALTMP}/output_*.txt ${PJM_O_WORKDIR}/${OUTPUT_DIR} ; \
#   fi'
```

ノード内テンポラリ領域の
データを第2階層ストレージ
にコピーする場合

ノード内テンポラリ領域の利用(2/2) [Fortran]

■ 第2階層ストレージのキャッシュ領域を介した出力の場合（デフォルトの出力）

◆ プログラムからの出力

```
NPROCS=      4  MYRANK=      0  DATA_SIZE=  2621440 . . .
  II=      5  ELAPSED(SEC)=  4.841709E-01
  II=     10  ELAPSED(SEC)=  3.833099E-01
. . .
  II=    100  ELAPSED(SEC)=  7.077831E-01
NPROCS=      4  TOTAL ELAPSED(SEC)=  8.782993E+00
```

◆ darshanの出力（抜粋）

<file_name>	<nprocs>	<slowest>	<avg>
/vol0006/.../output_000.txt	1	3.482252	3.482252
/vol0006/.../output_001.txt	1	5.392094	5.392094
/vol0006/.../output_002.txt	1	8.493428	8.493428
/vol0006/.../output_003.txt	1	6.423537	6.423537

■ ノード内テンポラリ領域に出力した場合

◆ プログラムからの出力

```
NPROCS=      4  MYRANK=      0  DATA_SIZE=  2621440 . . .
  II=      5  ELAPSED(SEC)=  8.438709E-02
  II=     10  ELAPSED(SEC)=  7.770016E-02
. . .
  II=    100  ELAPSED(SEC)=  7.657780E-02
NPROCS=      4  TOTAL ELAPSED(SEC)=  1.560476E+00
```

◆ darshanの出力（抜粋）

<file_name>	<nprocs>	<slowest>	<avg>
/local/output_000.txt	1	1.086248	1.086248
/local/output_001.txt	1	1.468742	1.468742
/local/output_002.txt	1	1.491861	1.491861
/local/output_003.txt	1	1.516526	1.516526

※出力ファイルサイズ: 20 MiB, ファイル出力回数: 20 (= 100 / 5), プロセス数: 4



共有テンポラリ領域の利用(1/2)

■ 共有テンポラリ領域を使用するジョブスクリプト例

```
#!/bin/sh
. . .
#PJM --llio sharedtmp-size=1Gi # 必要な共有テンポラリ領域のサイズを指定 (1ノード当たりのサイズ)
#PJM -L "node=4"
#PJM --mpi "max-proc-per-node=1"
#PJM -S
. . .
cat << EOD > ./input.txt
2621440    # 配列A のサイズ => ファイルサイズ2621440 Byte /1024/1024 * 8 = 20 MiB
100        # n_loop
5          # i_skip
${PJM_SHAREDTMP}/output.txt # ジョブ中で ${PJM_SHAREDTMP}は /share に展開される。
EOD
mpiexec -np ${PJM_MPI_PROC} -std-proc ./results/%J.log ${bindir}/run.x input.txt
```

double precision



共有テンポラリ領域の利用(2/2) [Fortran]

■ 第2階層ストレージのキャッシュ領域を介した出力の場合（デフォルトの出力）

◆ プログラムからの出力

```
NPROCS=      4  MYRANK=      0  DATA_SIZE=  2621440 . . .
   II=      5  ELAPSED(SEC)=  3.729321E-01
   II=     10  ELAPSED(SEC)=  3.361134E-01
. . .
   II=    100  ELAPSED(SEC)=  3.208321E-01
NPROCS=      4  TOTAL ELAPSED(SEC)=  6.873504E+00
```

◆ darshanの出力（抜粋）

```
# ****
# MPI-IO module data
# ****
. .
<file_name>          <nprocs>      <slowest>   <avg>
/vol0006/.../ output.txt      4        6.772257  6.739539
```

■ 共有テンポラリ領域に出力した場合

◆ プログラムからの出力

```
NPROCS=      4  MYRANK=      0  DATA_SIZE=  2621440 . . .
   II=      5  ELAPSED(SEC)=  1.034159E-01
   II=     10  ELAPSED(SEC)=  2.198033E-02
. . .
   II=    100  ELAPSED(SEC)=  1.922657E-02
NPROCS=      4  TOTAL ELAPSED(SEC)=  4.821570E-01
```

◆ darshanの出力（抜粋）

```
# ****
# MPI-IO module data
# ****
. .
<file_name>          <nprocs>      <slowest>   <avg>
/share/output.txt      4        0.479168  0.477262
```

※出力ファイルサイズ: 20 MiB, ファイル出力回数: 20 (= 100 / 5), ノード数: 4, プロセス数: 4



ハンズオンの内容 (1/2)

- 06_llio/01_localtmp, 06_llio/02_sharedtmp に取り組んでください。
 - ◆ まず、Readme.mdに目を通してください。

■ Exercise A:

- ◆ E1: ジョブスクリプトを確認して、ノード内テンポラリ領域、共有テンポラリ領域の使用方法を確認する。
- ◆ E2: ジョブを実行して、LLIOを利用した場合としなかった場合とで、出力にかかる時間を比較する。

■ Exercise Aが終わったら

- ◆ Exercise Bに取り組んでみてください。
 - darshan-parserを使ったファイルI/Oのプロファイル分析。
 - 出力ファイルサイズを変えた時の、ファイル出力にかかる時間の変化の調査。
 - ノード当たりのプロセス数を変えた時の、ファイル出力にかかる時間の変化の調査。

最後に

高度化支援のご案内



RISTは、「富岳」を中心とするHPCIシステムの利用研究課題を対象にアプリソフトの移植・高速化・高並列化等の支援を無償で実施中

■ 支援対象課題

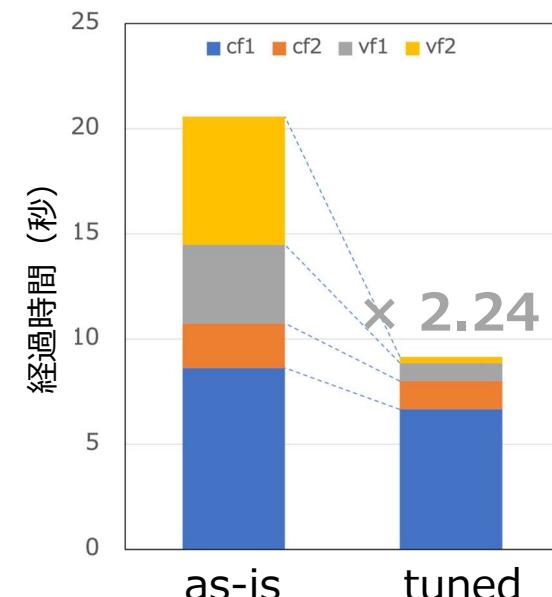
- 一般・若手課題、産業課題、臨時募集課題
 - 参考：https://www.hpci-office.jp/using_hpci/project_categories_overview
- 「富岳」成果創出加速プログラム課題

■ 支援内容

- 移植支援
 - 各計算機資源を実際に使用した動作確認等
- 高速化支援、高並列化支援
 - 実効性能分析（ホットスポット・ロードバランスの調査、プログラムの単体性能調査、並列性能調査）
 - 性能向上策の提案

■ 支援申込方法

- HPCIポータルサイトを参照
https://www.hpci-office.jp/user_support/tuning_support



支援例：FFVHC-ACEのカーネルプログラムに対する支援の効果
* 出典：https://www.hpci-office.jp/pages/e_meeting_A64FX_210427/#topic-2,

Appendix

Fugaku OnDemandの利用 (1/2)



富岳ウェブサイトのトップページ

スーパーコンピュータ「富岳」

運用状況
中規模ジョブ実行中
「富岳」運用ステータス
運用スケジュール

利用者支援
Open OnDemand
利用者ポータル
成果発表
申請
利用に関して
お問い合わせ

重要なお知らせ
2024-12-05 運用情報 ログインノード
2024-12-04 運用情報 認証ソフトウェア
2024-11-27 運用情報 Fortranプログラマ
2024-11-22 運用情報 ブリボスト環境
2024-11-20 運用情報 2024年度 Boost
お知らせ
2024-12-06 運用情報 【サテライト富岳】
2024-12-06 イベント 第45回「富岳」
2024-12-05 イベント ログインノード

注意: これはサンプルです。実際の画面と異なる可能性があります

Fugaku OnDemandのトップページ

Dashboard - Fugaku OnDemand

Fugaku OnDemand Batch Jobs Interactive Apps Passenger Apps

Welcome to the supercomputer Fugaku

RIKEN Center for Computational Science

Link

OnDemand Manual	
Fugaku Portal	
Fugaku	

Passenger Apps

- Active Jobs
- Budget Info
- Disk Info
- GakuNin RDM
- HPCI Shared Storage
- Fugaku Shell Access
- Home Directory

<https://ondemand.fugaku.r-ccs.riken.jp/pun/sys/dashboard>
(直接アクセスすることも可能)

Fugaku OnDemandの利用 (2/2)



最新の状態にする

terminalを開く

Refresh

Upload Download Copy/Move Delete

/ data / sample / Change directory Copy path

ディレクトリを変更する Show Owner/Mode Show Dotfiles Filter: Showing 4 rows - 0 rows selected

Type Name Modified at

Type	Name	Modified at
02viz.py	2024/12/6 18:02:36	
draw_graph.sif	2024/12/6 18:02:36	
main.f90	2024/12/6 18:02:36	
out.png	20.9 KB 2024/12/6 18:02:36	

View Edit Rename Download Delete

現在のディレクトリにあるファイルやディレクトリを表示

... ボタン

クリックする → ファイル/ディレクトリの操作のメニューが表示

注意: これはサンプルです。実際の画面と異なる可能性があります

if文付きループのSIMD処理 (02_simd-if): 計測例



■ 結果

◆ SIMD処理OFFからの速度向上([])を観察

- C++ clang: 低真率のKernel 1 → SIMD処理の効果が小さい
- Kernel 2 mod (リストベクトル作成): Kernel2と比較
→ 高真率で低下, 低真率で高速 (Fortran: -Ksimd=2との併用)

Legends

C++

- clang: Clang mode. Auto-vectorization with scalable vector length and interleaving are activated.
- clang.novec: Clang mode, without vectorization.

Fortran

- nosimd: Auto-vectorization is disable.
- simd: Auto-vectorization with -Ksimd=auto is activated.
- simd2: Auto-vectorization with -Ksimd=2 is activated.

Rate of "TRUE" in if stat. = 0.86	Elapsed time (sec.) [C++]		Elapsed time (sec.) [Fortran]		
	clang.novec	clang	nosimd	simd	simd2
Kernel 1	1.090	0.352 [x3.1]	1.062	0.157 [x6.8]	0.154 [x6.9]
Kernel 2	1.079	0.174 [x6.2]	1.125	0.182 [x6.2]	0.182 [x6.2]
Kernel 2 mod	0.976	0.349 [x2.8]	1.057	0.354 [x3.0]	0.202 [x5.2]

Rate of "TRUE" in if stat. = 0.29	Elapsed time (sec.) [C++]		Elapsed time (sec.) [Fortran]		
	clang.novec	clang	nosimd	simd	simd2
Kernel 1	0.467	0.351 [x1.3]	0.466	0.154 [x3.0]	0.154 [x3.0]
Kernel 2	0.427	0.174 [x2.5]	0.487	0.182 [x2.7]	0.182 [x2.7]
Kernel 2 mod	0.388	0.171 [x2.3]	0.469	0.257 [x1.8]	0.120 [x3.9]

※ブーストエコモードでの測定 (freq=2200, eco_state=2)

DGEMMの計測 (02_mm-vs-mv) (1/5)



■ 密行列の行列行列積 (行列サイズ ~ N)

- ◆ 演算量~ $O(N^3)$, メモリアクセス~ $O(N^2)$ → キャッシュの利用, SIMD処理の利用等で性能向上が期待
- ◆ 実際は: “Well-tuned”なライブラリを利用すべき
 - 各種最適化 (ブロッキング, アンローリング, ベクトル化, スレッド並列化, etc.)が適用済み
 - 特に「大きな」サイズでは利用を検討

■ カーネル: 3種類を検討 (すべて同じ計算)

- ◆ simple: 自前で書いた行列行列積
 - 最適化はコンパイラ任せ (-O3)
- ◆ DGEMM: 実行列の行列行列積ルーチン
- ◆ Rep-DGEMV: 行列ベクトル積 (DGEMV) の繰り返しで行列行列積を計算
 - DGEMMからブロッキングなどの最適化を落とした実装とみなすことができる

$$\begin{aligned} C &\leftarrow C + A \times B \\ \Leftrightarrow \\ (\vec{c}_1, \dots, \vec{c}_N) &\leftarrow (\vec{c}_1, \dots, \vec{c}_N) + A \times (\vec{b}_1, \dots, \vec{b}_N) \end{aligned}$$



DGEMMの計測 (02_mm-vs-mv) (2/5) [Fortran]

```
! Code fraction (main.F90)
do it = 1, NITR
    call mmp_simple(matc, mata, matb, NSIZE)
    matc(1,1) = dble(it)*1.0E-9_DP
end do

do it = 1, NITR
    call DGEMM('N', 'N', NSIZE, NSIZE, NSIZE, one, mata, &
&           NSIZE, matb, NSIZE, zero, matc, NSIZE)
    matc(1,1) = dble(it)*1.0E-9_DP
end do

incx = 1
incy = 1
do it = 1, NITR
    do j = 1, NSIZE
        call DGEMV('N', NSIZE, NSIZE, one, mata, &
&           NSIZE, matb(:,j), incx, zero, matc(:,j), incy)
    end do
    matc(1,1) = dble(it)*1.0E-9_DP
end do
```

```
! Code fraction (mykernel.F90)
subroutine mmp_simple (MC, MA, MB, ns)

    do j = 1, ns
        do k = 1, ns
            do i = 1, ns
                MC(i,j) = MC(i,j) + MA(i,k)*MB(k,j)
            end do
        end do
    end do
```



DGEMMの計測 (02_mm-vs-mv) (2/5) [C]

```
! Code fraction (main.c)
#include "cblas.h"

for ( int it = 0; it < NITR; ++it) {
    mmp_simple (NSIZE, matc, mata, matb);
    matc[0] = (double)it * 1.0e-9;
}

for ( int it = 0; it < NITR; ++it) {
    cblas_dgemm (CblasRowMajor, CblasNoTrans,CblasNoTrans,
        NSIZE, NSIZE, NSIZE,one, &mata[0], NSIZE, &matb[0],
        NSIZE, zero, &matc[0], NSIZE);
    matc[0] = (double)it * 1.0e-9;
}

for ( int it = 0; it < NITR; ++it) {
    for ( int j = 0; j < NSIZE; ++j ) {
        /* pack buffer: vx <- matb */
        cblas_dgemv (CblasRowMajor,CblasNoTrans,NSIZE,NSIZE,
            one, &mata[0],NSIZE,&vx[0],incx,zero,&vy[0],incy);
        /* unpack buffer: matc <- vy */
    }
    matc[0] = (double)it * 1.0e-9;
}
```

```
! Code fraction (mykernel.c)
void mmp_simple (const int ns, double * restrict mc,
double * restrict ma, double * restrict mb)
{

    for ( i = 0; i < ns; ++i ) {
        for ( k = 0; k < ns; ++k ) {
            int ki = k + ns*i;
            for ( j = 0; j < ns; ++j ) {
                int ji = j + ns*i;
                int jk = j + ns*k;
                mc[ji] += ma[ki]*mb[jk];
            }}}
```



■ 設定

- ◆ データサイズ: 最大で44.8MiB (> L2キャッシュ全体)
 - 実正方形行列の次元を20 - 1400まで変動
- ◆ 時間計測: 各カーネルを (最大で) 10000回繰り返し実行し, 経過時間を測定
 - 次元の大きい場合は繰り返し回数を減少させる
 - [(演算量) × (繰り返し回数)] / (経過時間) でFlop/sを評価
- ◆ 使用するライブラリ: 富士通SSL2 (富士通版のBLAS/LAPACKを使用)
 - DGEMM, DGEMVで使用

■ ポイント

- ◆ 行列サイズ (正方形行列の次元) を大きくしたときの性能 (GFlop/s) の挙動
- ◆ DGEMMとRep-DGEMVの性能の差



DGEMMの計測 (02_mm-vs-mv) (4/5)

■ ログの出力例 (outfile.*)

- ◆ DGEMMがもっとも速い (Rep-DGEMVの4-5倍速い)

```
# 行列サイズ20 (outfile.20)
kernel NSIZE NITR Elapsed_time_sec Gflop/s (省略)
    simple      20      10000      0.0850      1.8819 ...
    DGEMM       20      10000      0.0075     21.4330 ...
    Rep-DGEMV   20      10000      0.0357      4.4865 ...
```

繰り返し回数
(NITR)

経過時間 (sec.)

GFlop/s

```
# 行列サイズ1200 (outfile.1200)
kernel NSIZE NITR Elapsed_time_sec Gflop/s (省略)
    simple      1200      5      1.7403      9.9294 ...
    DGEMM       1200      5      0.2888     59.8337 ...
    Rep-DGEMV   1200      5      1.2972     13.3210 ...
```

DGEMMの計測 (02_mm-vs-mv) (5/5) [Fortran]



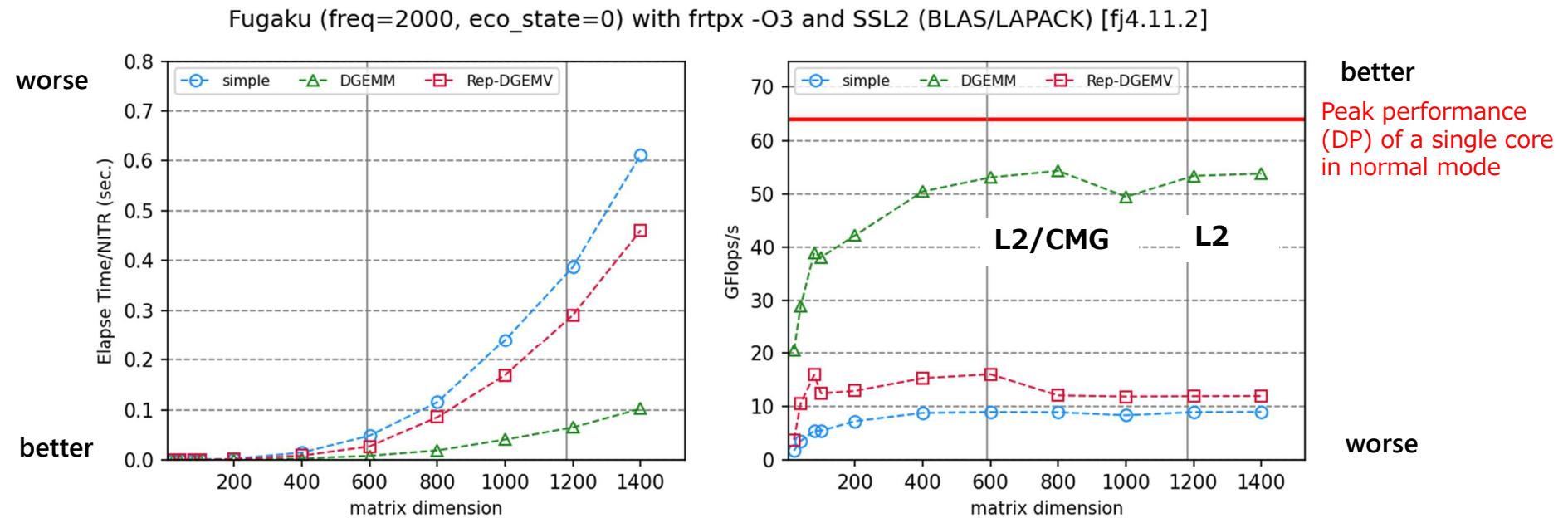
■ 結果 [Fortran; 通常モード]

◆ 測定した範囲の行列サイズ全般でDGEMMの優位性を確認

- large size (L2/CMG以上), small size(行列サイズ)<100)のどちらでもDGEMMが優位
- DGEMMの性能の飽和値 (行列サイズ→ large): ~54Gflop/s → ピーク性能(DP)の84%程度

◆ ライブラリ利用のtips

- できるだけ大きなサイズの行列をDGEMMで処理
- MVをMMにまとめあげることができるか検討



DGEMMの計測 (02_mm-vs-mv) (5/5) [Fortran]



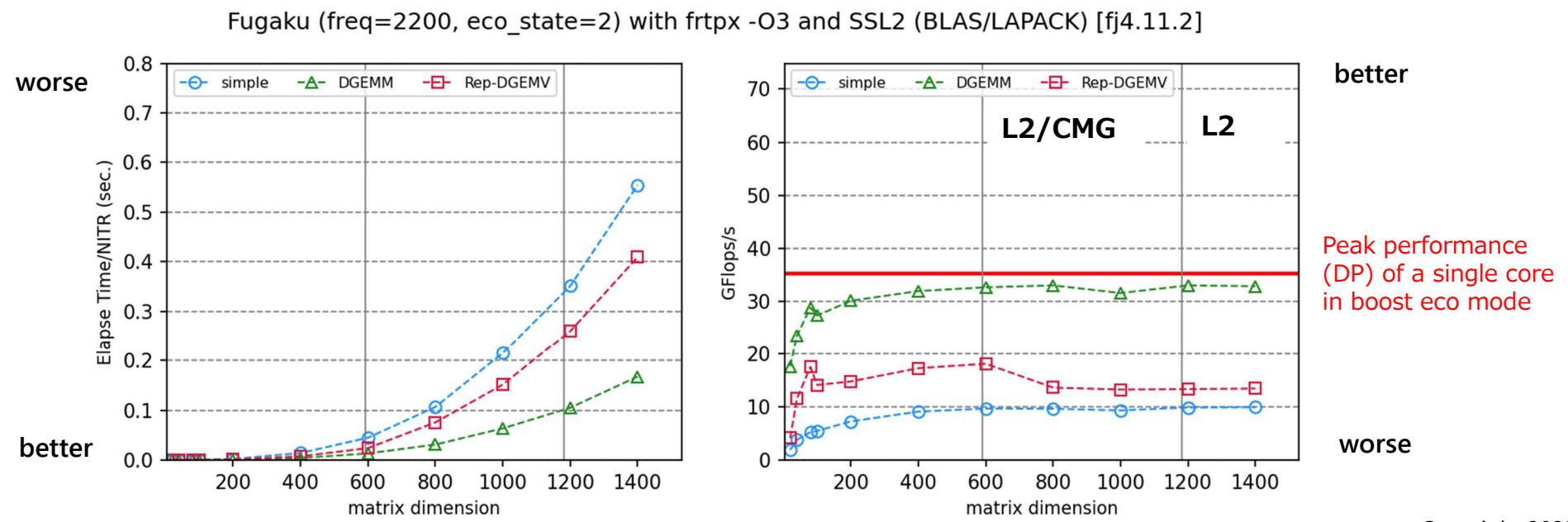
■ 結果 [Fortran; ブーストエコモード]

◆ 測定した範囲の行列サイズ全般でDGEMMの優位性を確認

- large size (L2/CMG以上), small size(行列サイズ)<100)のどちらでもDGEMMが優位
- 2本あるpipelineの内、1本しか使っていないため、DGEMMの性能は32 GFlops/sで頭打ち

◆ 通常モードとブーストエコモードの選択

- 短いテスト計算で、どちらのモードが良いか事前に検討することが重要

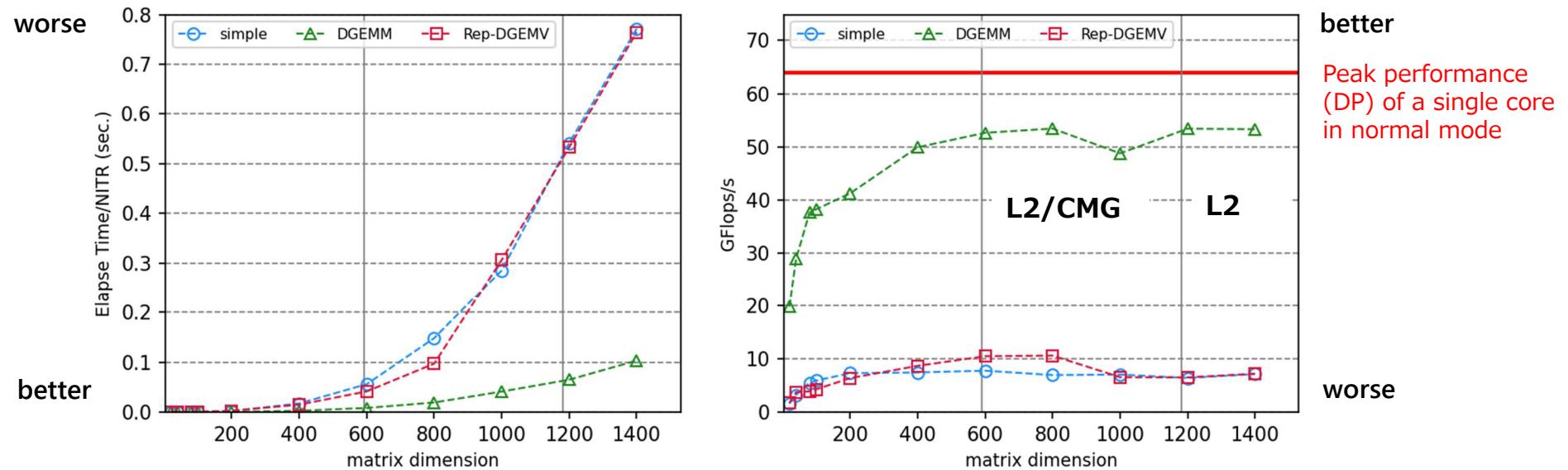


DGEMMの計測 (02_mm-vs-mv) (5/5) [C]

■ 結果 [C; 通常モード]

- ◆ 測定した範囲の行列サイズ全般でDGEMMの優位性を確認
 - large size (L2/CMG以上), small size(行列サイズ)<100)のどちらでもDGEMMが優位
 - DGEMMの性能の飽和値 (行列サイズ→ large): ~53Gflop/s → ピーク性能(DP)の83%程度
- ◆ ライブラリ利用のtips
 - できるだけ大きなサイズの行列をDGEMMで処理
 - MVをMMにまとめあげることができるか検討

Fugaku (freq=2000, eco_state=0) with fccpx -Nclang -O3 -ffj-interleave-loop-insns and SSL2 (BLAS/LAPACK)[fj4.11.2]

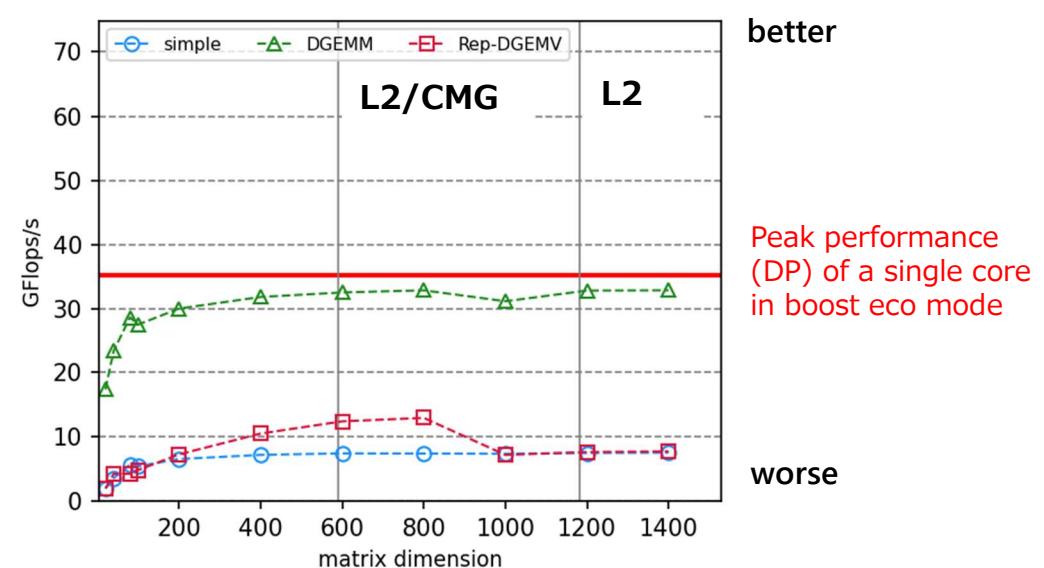
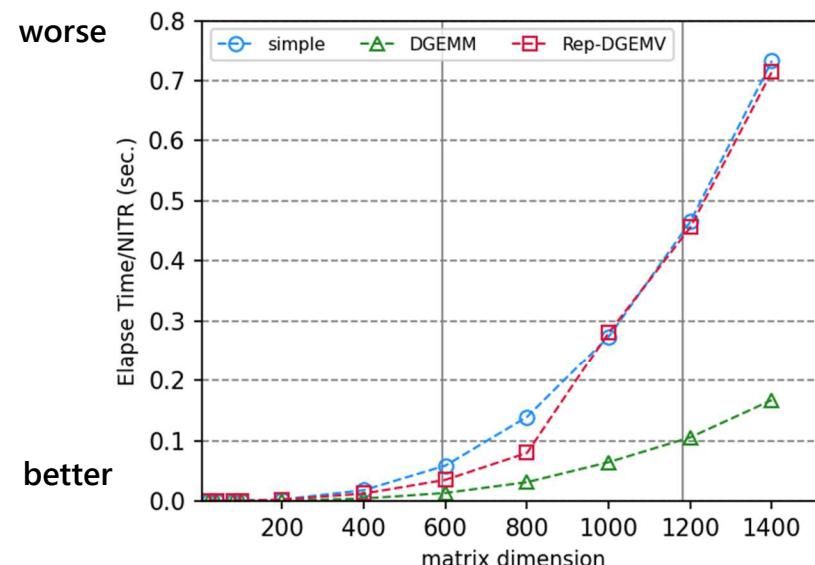


DGEMMの計測 (02_mm-vs-mv) (5/5) [C]

■ 結果 [C; ブーストエコモード]

- ◆ 測定した範囲の行列サイズ全般でDGEMMの優位性を確認
 - large size (L2/CMG以上), small size(行列サイズ)<100)のどちらでもDGEMMが優位
 - 2本あるpipelineの内、1本しか使っていないため、DGEMMの性能は32 GFlops/sで頭打ち
- ◆ 通常モードとブーストエコモードの選択
 - 短いテスト計算で、どちらのモードが良いか事前に検討することが重要

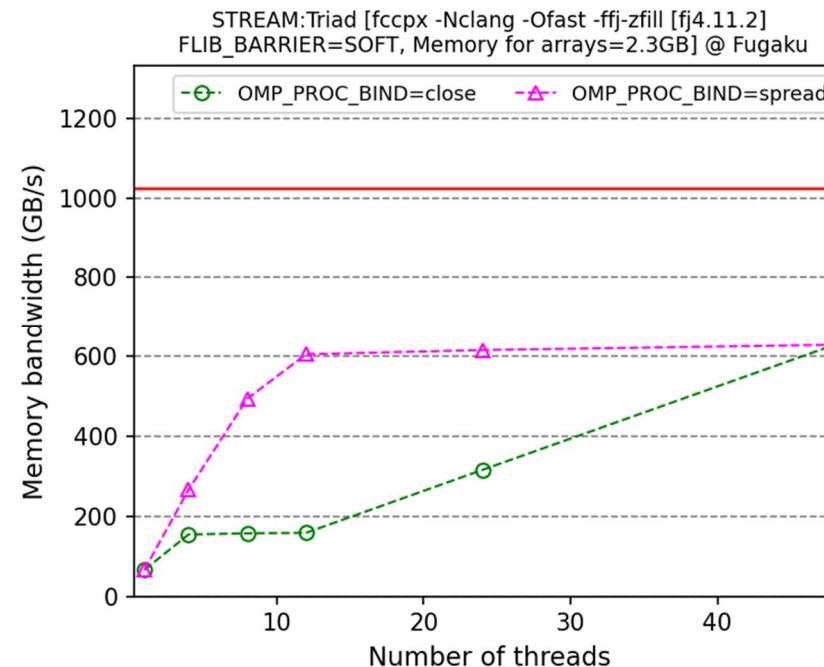
Fugaku (freq=2200, eco_state=2) with fccpx -Nclang -O3 -ffj-interleave-loop-insns and SSL2 (BLAS/LAPACK)[fj4.11.2]



Thread affinity (02_thread-affinity): C Clang modeの挙動

■ [試行] C Clang modeでの挙動

- ◆ >12スレッド: FortranやC Tradと同様な挙動
 - spreadに比べcloseで顕著な性能劣化
- ◆ <=12スレッド: spreadでメモリバンド幅が大幅に上昇(?)
 - Fortran, C Tradとの違いの理由が不明
 - GOMP_CPU_AFFINITYでより詳細に設定しても同じ傾向
 - runtimeはFortran, C Tradと同じLLVM (-Nlibomp)



Data locality (03_data-locality): 計測例



■ 結果

◆ Paging policyの設定によるメモリバンド幅の変化

- 未指定 (default)はdemand:demand:prepageと推測
- 高スレッド (48スレッド) ほど影響が大きい

Fortran (frtpx -Kfast -Kzfill): 48 threads

Paging policy	OMP_PROC_BIND	STREAM: Triad (MB/s)
未指定 (default)	spread	810754.64
P:D:P	spread	105707.67
D:D:P	spread	810819.94

D=demand, P=prepage

Fortran (frtpx -Kfast -Kzfill): 12 threads

Paging policy	OMP_PROC_BIND	STREAM: Triad (MB/s)
未指定 (default)	close	205884.88
P:D:P	close	129038.96
D:D:P	close	205443.68

point-to-point通信の基礎性能 (01_pingpong): 計測例



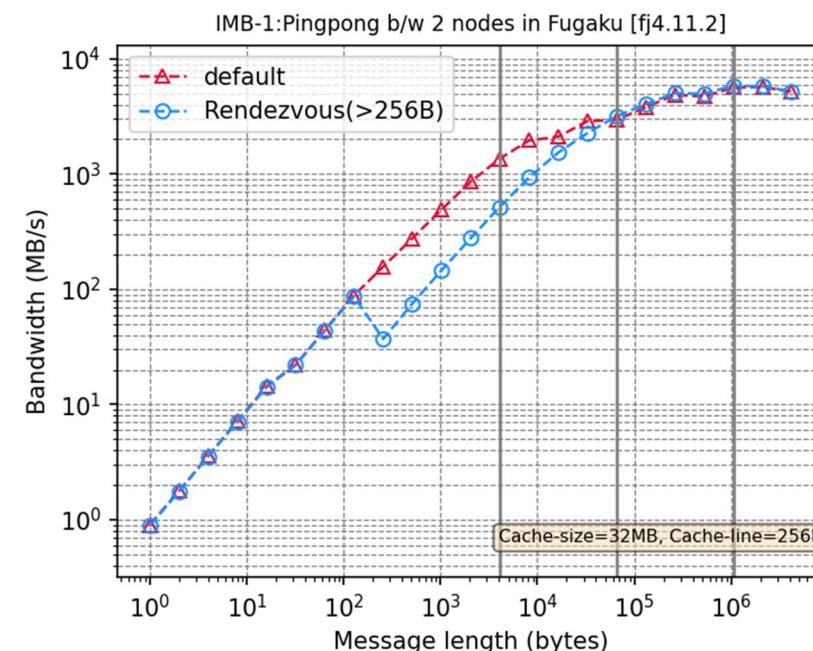
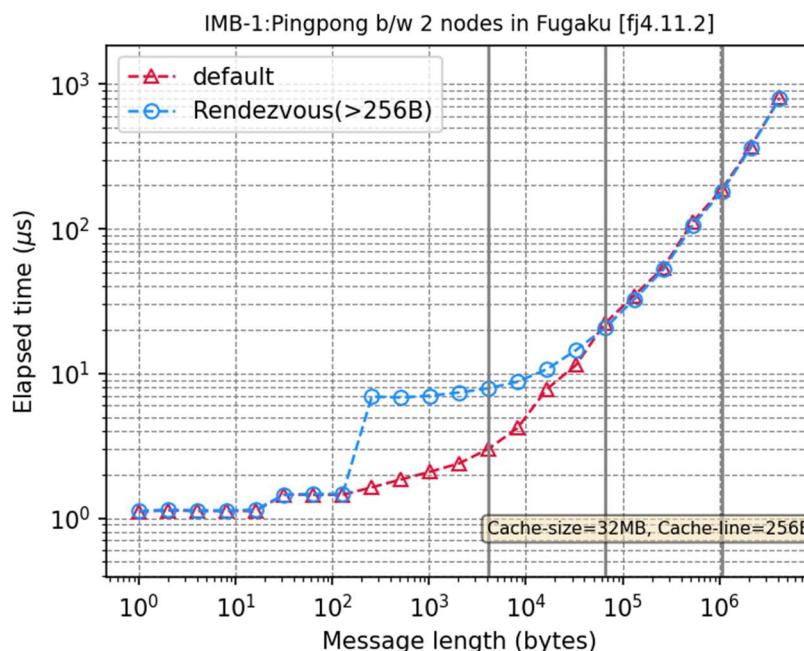
■ 結果

- ◆ IMB (Intel MPI benchmark)のPingpong (Send, Recv) による2ノード間計測

(Intel MPI benchmark: <https://github.com/intel/mpi-benchmarks.git>)

- default: Eager通信から Rendezvous通信への自動切り替え
- Rendezvous: メッセージ長が256バイト以下を除き、常にRendezvous通信

メモ
ジョブスクリプトの設定
• #PJM -L "node=2"
• #PJM --mpi "rank-map-bynode"
• #PJM --mpi "max-proc-per-node=2"



Multiple PingPong (02_mpingpong): 計測例



■ 結果

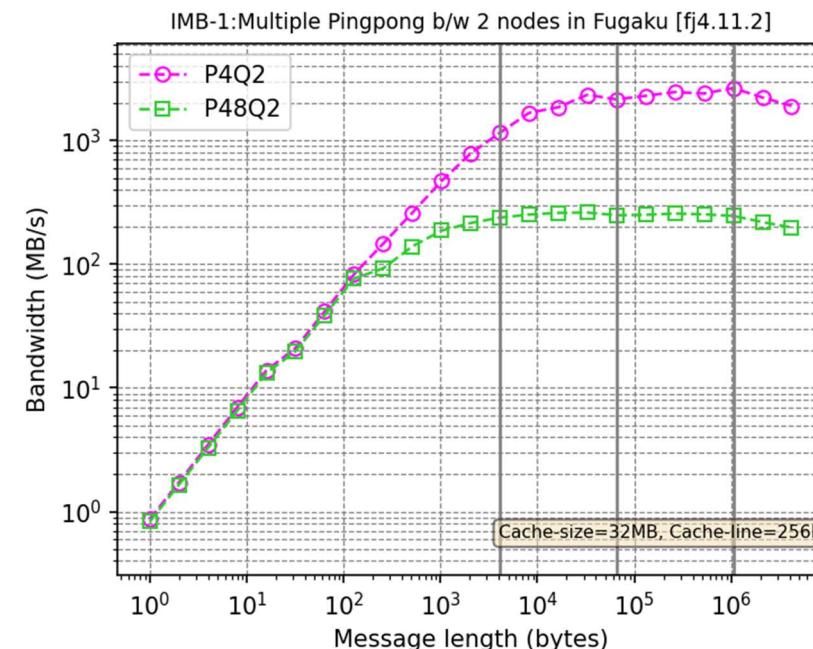
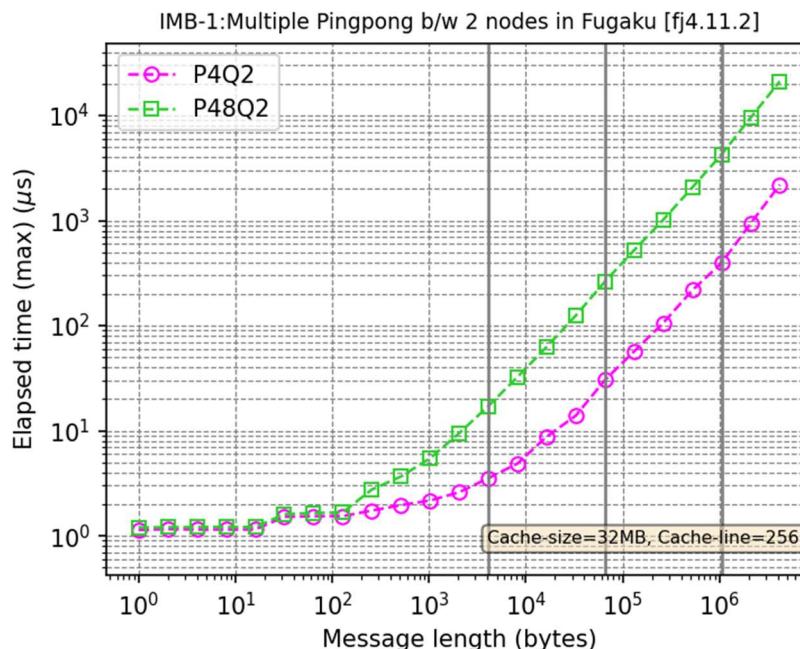
- ◆ IMB (Intel MPI benchmark)のPingpongによる2ノード間計測

(Intel MPI benchmark: <https://github.com/intel/mpi-benchmarks.git>)

- P4Q2: PPN=4で2ノード間のPingpongを多重実行
- P48Q2: PPN=48で2ノード間のPingpongを多重実行

メモ

- ジョブスクリプトの設定
 - #PJM -L "node=2"
 - #PJM --mpi "rank-map-bychip"



vcoordfileを使ったランク割当 (01_vcoord): 出力例



■ 結果

- ◆ 1次元座標 (形状指定なし) によるvcoordfile(vcoord)付きでランク割当とコアバインディングを制御
 - 各ランクの標準出力に割当情報を出力して確認 (ホスト名とnumactl --showの出力)

```
# MPIジョブの実行
## vcoordfileの行数を取得→MPIプロセス数に設定
NPROCS=$(awk 'END{print NR}' vcoord)
## vcoordfile付きで実行
mpiexec -np ${NPROCS} --vcoordfile vcoord \
--stdout-proc output ... bash run.sh ...
```

vcoordの中身

```
(0) core=1 ← ランク0をノード(0)に割当, スレッド数は1
(1) core=1
(1) core=12
(1) core=12 ← ランク3をノード(1)に割当, スレッド数は12
(1) core=12
```

1次元座標:
5行
→ 5プロセス

```
# run.sh
hostname
numactl --show #プログラム($1)の前に実行
$1 #スクリプト引数をプログラムとして受け取り実行
```

```
# output.*.0 (ランク0の標準出力)
# numactl --showの出力のみ表示
```

```
...
physcpubind: 12 ← バインドされたコアのIDは12 (1スレッド)
cpubind: 4
nodebind: 4 ← CMGのIDは4
membind: 4 5 6 7
```

```
# output.*.3 (ランク3の標準出力)
# numactl --showの出力のみ表示
```

```
...
physcpubind: 36 37 38 39 40 41 42 43 44 45 46 47 ← バインドされたコアのIDは36-47 (12スレッド)
cpubind: 6
nodebind: 6 ← CMGのIDは6
membind: 4 5 6 7
```



- ◆ Fujitsu github
 - <https://github.com/fujitsu>
- ◆ Meeting for application code tuning on A64FX computer systems
 - https://www.hpci-office.jp/en/events/symposia/meetings_A64FX
- ◆ A64FX Tuning Documents;
 - https://github.com/RIKEN-RCCS/A64FX_Tuning_Documents
- ◆ 高速化ノウハウ集 (HPCIポータル)
 - https://www.hpci-office.jp/user_support/tuning_knowhow
 - 「富岳」の利用について事例が掲載