東京大学
THE UNIVERSITY OF TOKYO

東京大学情報基盤センター
INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

第205回 お試しアカウント付き 並列プログラミング講習会 「Wisteria実践」

東京大学情報基盤センター
塙　敏博
三木 洋平

講習会概略

- **開催日：**
  2023年6月8日（木）　10：30 - 17：45
- **形態：**ZoomとSlackを用いたオンライン講習会
- **講習会プログラム：** 講師：塙
  - 10：30 – 12：00 **Wisteria/BDEC-01**システム紹介
  - 13：30 – 15：15 **Odysseyノード**：A64FXにおけるOpenMP並列化、MPI+OpenMPハイブリッド並列、性能分析（講義+演習）
  - 15：30 - 17：15 **Aquariusノード:** A100 GPUの利用、OpenACC、MPI並列化、性能分析（講義+演習）
  - 17：30 –　　Q&A

お願い等

- ハンズオンのためのPC，Zoom及びスパコンへ接続するためのネットワーク環境は各受講者でご準備ください。
- PCは Windows/Microsoft Update，Apple Security Updateなどで最新のセキュリティアップデートを行ってください。
- 必ずウィルス対策ソフトウェアをインストールし，ウィルス検索を実行して問題がないことを事前に確認してから受講してください。
  - セキュリティ対策未実施の場合はオンライン講習会受講を認めません。
- OSは、Windows、Macどちらでも構いませんが、SSHを用いてセンターのスーパーコンピューターへ接続ができることが必要です（後述）。
- 演習の実施に当たり，受講生にセンターのスーパーコンピューターを1月間利用できる無料アカウント（お試しアカウント）を発行します。

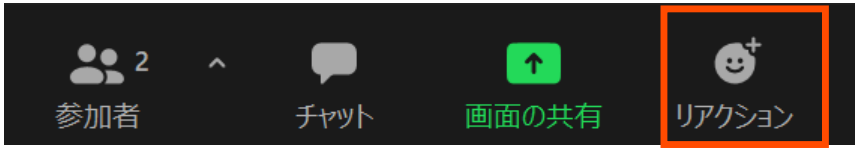
Zoom関連

- 「手をあげる」機能
  - 質問がある際，全体の状況を確認するため使用
- ブレークアウトセッション
  - 画面を共有しながらエラー対応する際に使用
  - （なるべく口頭でのやりとりやSlackで対応する予定）

- https://utelecon.adm.u-tokyo.ac.jp/zoom/how_to_use

「手を挙げる」方法

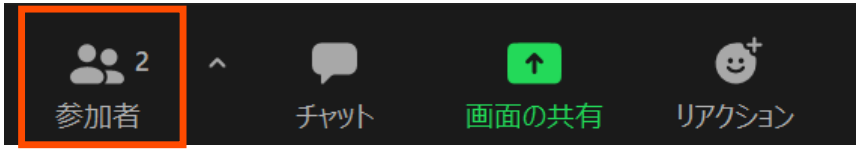
1．Zoomメニュー中の「リアクション」をクリック



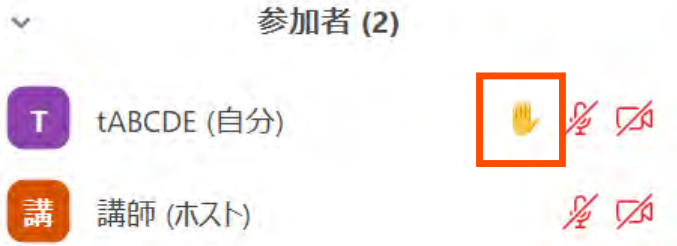
2．ポップアップで表示された「手を挙げる」をクリック

手が挙がっていることの確認方法

1. Zoomメニュー中の「参加者」をクリックして，参加者一覧を表示

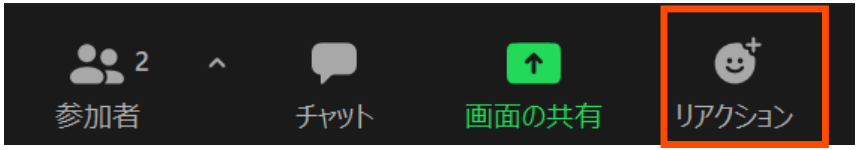


2. 表示された参加者一覧の，自分のところを見ると手が挙がっている



「手を降ろす」方法

1．Zoomメニュー中の「リアクション」をクリック



2．ポップアップで表示された「手を降ろす」をクリック

ブレイクアウトルーム（1/6）

- 演習時に使用するかもしれません
- 演習中に「ヘルプを求める」ことができます
  - ホストを招待した後に「画面を共有」することで，皆さんの記述したプログラムを一緒に見ながら問題解決にあたります

- Zoomメニュー中の「ブレイクアウトルーム」をクリック

ブレイクアウトルーム（2/6）

- 進行中のブレイクアウトルームのリストが表示されるので，空いている部屋に「参加」してください
  - 左の例では5部屋がすべて空室，右の例ではルーム1のみ参加者がいる

ブレイクアウトルーム（3/6）

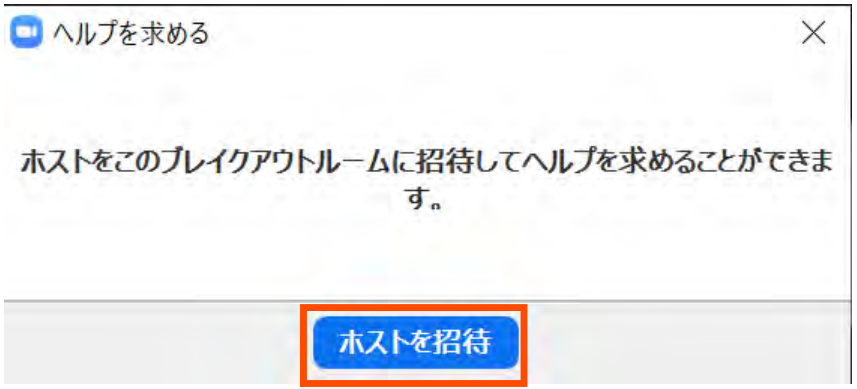
- 「参加」をクリックすると確認画面が出てくるので，「はい」を選択すると入室できます

ブレイクアウトルーム（4/6）

- 再度メニュー中の「ブレイクアウトルーム」をクリックすると，「ヘルプを求める」が増えているのでクリックしてください

ブレイクアウトルーム（5/6）

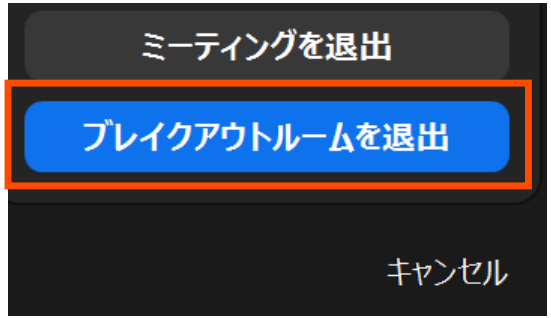
- ポップアップで出てきた「ホストを招待」をクリック
- ホスト（講師）の参加待ちに移行します（画面はそのまま）
  - 他の受講者のヘルプ中など，直ちに対応できない場合もあります



ブレイクアウトルーム（6/6）

- 問題解決後は，Zoomメニュー中の「ルームを退出する」

- 「ブレイクアウトルームを退出」が表示されるのでクリックして，元の講習会会場にお戻りください
  - 間違えて「ミーティングを退出」すると講習会から退出します

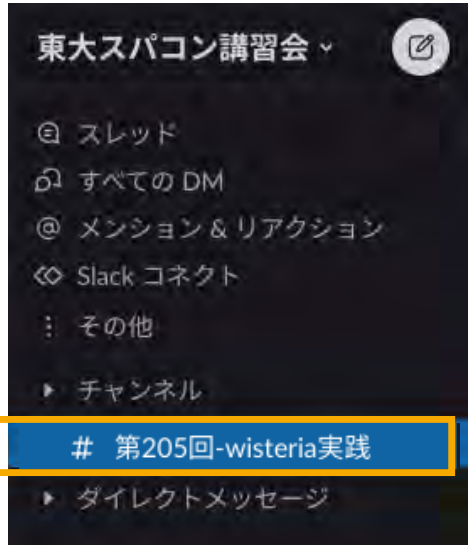


Slack関連

- ブラウザ上で使う場合には：
  - https://w1590055008-bgo338004.slack.com/archives/C05ASCUE293
  - 注：ログインには，事前にお配りしたリンクからの登録が必要です
- 質問対応に使用
- コードの貼り付け方
- スレッドの確認方法

- 以下，ブラウザ版で説明しますがアプリ版でも操作は同じです

質疑応答チャンネルへの移動

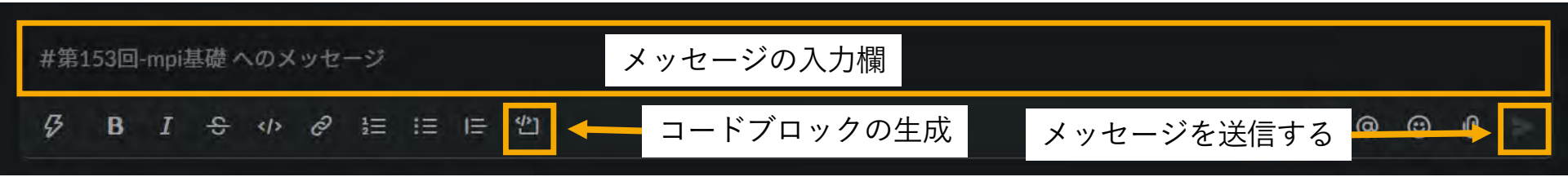


- 左側のメニューバーのチャンネル一覧内に「第170回-」があるので，クリック

- 表示されていない場合
  1. 「チャンネルを追加する」をクリック
  2. 「チャンネル一覧を確認する」をクリック
  3. 「第205回-wisteria実践」があるので，「参加する」をクリック
     - 第199回～それ以前とお間違えなく

メッセージの入力方法

- 最下部に入力欄があるので，質問内容を記載して Ctrl+Enter
  - 入力後に右下の「メッセージを送信する」をクリックしても同じ
    （メッセージ入力前には，「メッセージを送信する」は押せない）



- コードを入力する際には，「コードブロック」がおすすめ
  - 枠が生成されるので，この中にコピペするのが簡単かつ見やすい
  - ```（JIS配列ならばShift+@を3連打）しても枠が生成される

自分が参加したスレッドのみの表示方法

- 左上の「スレッド」をクリックすると，自分が参加しているスレッドの一覧が表示されます
  - 質問内容には，「スレッドで返信する」形式で回答するので，自分が聞いた内容のみが表示できます

Wisteria/BDEC-01の紹介

別資料参照

参考：ジョブクラス：インタラクティヴジョブ（トークン消費無し）

https://www.cc.u-tokyo.ac.jp/supercomputer/wisteria/service/job.php

- Wisteria-O（Odyssey）：シミュレーションノード群

キュー名	ノード数	制限時間（Elapsed）	メモリ容量（GB）
interactive-o (interactive-o_n1) (interactive-o_n12)	 1 2 ～ 12	 30 分 10 分	 28 28

- Wisteria-A（Aquarius）：データ・学習ノード群

キュー名	ノード数	制限時間（Elapsed）	メモリ容量（GB）
interactive-a	1ノード	10 分	448
share-interactive	1GPU	10 分	56

参考：インタラクティブ実行のやり方（本講習会では使えません）

Odyssey

- 1ノード実行

```
$ pjsub --interact -g グループ名 -L rg=interactive-o,elapse=01:00
```

- 12ノード実行

```
$ pjsub --interact -g グループ名 -L rg=interactive-o,node=12,elapse=01:00
```

Aquarius

- 1GPU実行

```
$ pjsub --interact -g グループ名 -L rg=share-interactive,elapse=01:00
```

- 1ノード(8GPU)実行

```
$ pjsub --interact -g グループ名 -L rg=interactive-a,elapse=01:00
```

※インタラクティブ用のノードがすべて使われている場合、資源が空くまでログインできません。
※このアカウントでは使えません。

参考：ジョブクラス：バッチジョブ（Odyssey）

https://www.cc.u-tokyo.ac.jp/supercomputer/wisteria/service/job.php

キュー名	ノード数	制限時間（Elapsed）	メモリ容量（GB）
debug-o	1 ～ 144	30 分	28
short-o	1 ～ 72	8 時間	28
regular-o			
(small-o)	1 ～ 144	48 時間	28
(medium-o)	145 ～ 576	〃	〃
(large-o)	577 ～ 1152	〃	〃
(x-large-o)	1153 ～ 2304	24 時間	〃
priority-o （優先実行，トークン消費量 1.5倍）	1 ～ 288	48 時間	28

参考：ジョブクラス：バッチジョブ（Aquarius）

https://www.cc.u-tokyo.ac.jp/supercomputer/wisteria/service/job.php

キュー名	ノード数 GPU数	制限時間 （Elapsed）	メモリ容量 （GB）
debug-a	1 ノード	30 分	448
short-a	1 ～ 2 ノード	2 時間	448
regular-a 　(small-a) 　(medium-a) 　(large-a)	 1 ～ 2 ノード 3 ～ 4 ノード 5 ～ 8 ノード	 48 時間 〃 24 時間	 448 〃 〃
share-debug	1, 2, 4 GPU	30 分	56
share-short	1, 2, 4 GPU	2 時間	56
share 　(share-1) 　(share-2) 　(share-4)	 1 GPU 2 GPU 4 GPU	 48 時間 〃 24 時間	 56 〃 〃

参考：ジョブクラス：プリポストサービス
ログインノードと同じアーキテクチャのノードを使用

https://www.cc.u-tokyo.ac.jp/supercomputer/wisteria/service/job.php

キュー名	ノード数	制限時間（Elapsed）	メモリ容量（GB）
prepost（予約無し）	1	6 時間	340
prepost1_n1 ～ prepost4_n1	1	1 ～ 6 時間	340
prepost1_n4	1 ～ 4	1 ～ 6 時間	340
prepost1_n8	1 ～ 8	1 ～ 6 時間	340

教育利用のジョブクラス：バッチジョブ（Odyssey）

キュー名	ノード数	制限時間（Elapsed）	メモリ容量（GB）
lecture-o （教育利用全体で共有）	1 ～ 12	15 分	28
tutorial-o （この講習会の時間中のみ使用可能）	1 ～ 12	15 分	28

ノード当たり 48コア ➔ 最大576コアまで使用可能！

教育利用のジョブクラス：バッチジョブ（Aquarius）

キュー名	GPU数	制限時間（Elapsed）	メモリ容量（GB）
lecture-a （教育利用全体で共有）	1,2,4	15 分	56/GPU
tutorial-a （この講習会の時間中のみ使用可能）	1,2,4	15 分	56/GPU

share-aと同様の使い方

「Wisteria実践」について

本講習会の目的

並列アルゴリズムはひとまず置いといて…

- Odysseyノードの多数のコアの扱い方
- Odysseyの多数の計算ノードによる効率の良い実行方法
  - 富士通コンパイラ、ジョブ環境のTIPS
- AquariusノードのGPUの扱い方

等について学ぼう！

（現代の）スパコンは並列性が命

- 京コンピュータ(2011~2019)： 88,128ノード x 8コア/ノード
- Oakforest-PACS : 8,208ノード x 68コア/ノード
- Sunway TaihuLight: 40,960ノード x 260コア/ノード
- Summit : 約4,600ノード x 6 GPU/ノード
- 富岳: 158,976ノード x 48コア/ノード
- Frontier: 9,248ノード x (64コア + 4 GPU)/ノード
- ABCI: 1,088ノードx 4 GPU/ノード+120ノード x 8 GPU/ノード
- Wisteria: 7,680ノード x 48コア/ノード + 45ノード x(72コア+8 GPU)/ノード

- 1コア当たりではスパコンの方が遅いこともある
  - スパコン向けは，電力効率を狙ってクロック周波数を下げている
    ➔ その分コア数を増やす

並列プログラムの実行主体

- マルチスレッド
  - OpenMP
    - ユーザが並列化指示行を記述
- マルチプロセス
  - MPI （Message Passing Interface）
    - ユーザがデータ分割方法を明示的に記述

共有メモリで動作
→並列化の指示だけでOK

- ノードが物理的に異なる
- 同一ノードでもプロセス毎にメモリは論理的に分離
  →明示的な通信が必要

マルチプロセスとマルチスレッドは共存可能
→ハイブリッドMPI/OpenMP実行

Weak ScalingとStrong Scaling

MPI通信もポイント
Odyssey: 低レイテンシ
Aquarius: 高バンド幅

並列処理においてシステム規模を大きくする方法

- Weak Scaling: それぞれの問題サイズは変えず並列度をあげる
  - ➢全体の問題サイズが（並列数に比例して）大きくなる
  - ➢通信のオーバヘッドはあまり変わらないか、やや増加する
- Strong Scaling: 全体の問題サイズを変えずに並列度をあげる
  - ➢問題サイズが装置数に反比例して小さくなる
  - ➢通信のオーバヘッドは相対的に大きくなる

Weak Scaling
それまで解けなかった規模の問題が解ける

Strong Scalingも重要
同じ問題規模で、短時間に結果を得る（より難しい）

並列プログラミングについて学習したい方へ

以下の講習会資料を参考にしてください（初級➔上級）

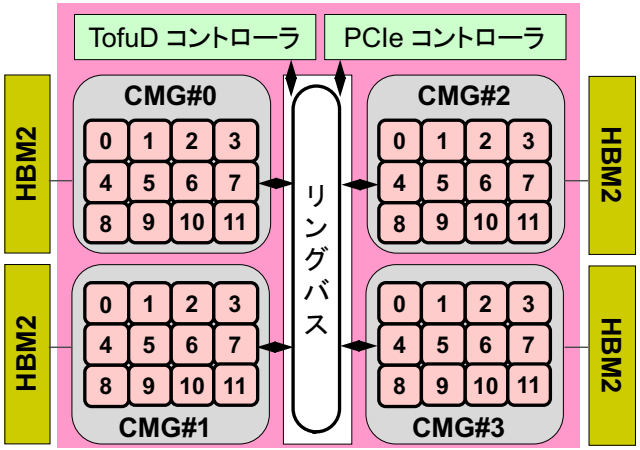
https://www.cc.u-tokyo.ac.jp/events/lectures/

- 「MPI基礎：並列プログラミング入門」
  - https://www.cc.u-tokyo.ac.jp/events/lectures/203/
- 「OpenACCとMPIによるマルチGPUプログラミング入門」
  - https://www.cc.u-tokyo.ac.jp/events/lectures/209/
- 「OpenMPによるマルチコア・メニィコア並列プログラミング入門」
  - https://www.cc.u-tokyo.ac.jp/events/lectures/192/
- 「GPUプログラミング入門」
  - https://www.cc.u-tokyo.ac.jp/events/lectures/206/
- 「並列有限要素法で学ぶ並列プログラミング徹底入門」
  - https://www.cc.u-tokyo.ac.jp/events/lectures/194/
- 「OpenMPで並列化されたC++プログラムのGPU移植手法」
  - https://www.cc.u-tokyo.ac.jp/events/lectures/208/
- 「MPI+OpenMPで並列化されたFortranプログラムのGPUへの移行手法」
  - https://www.cc.u-tokyo.ac.jp/events/lectures/211/

「こんな講習会が欲しい！」という要望があればぜひアンケートへ！

Wisteria-Odyssey
A64FXの基礎

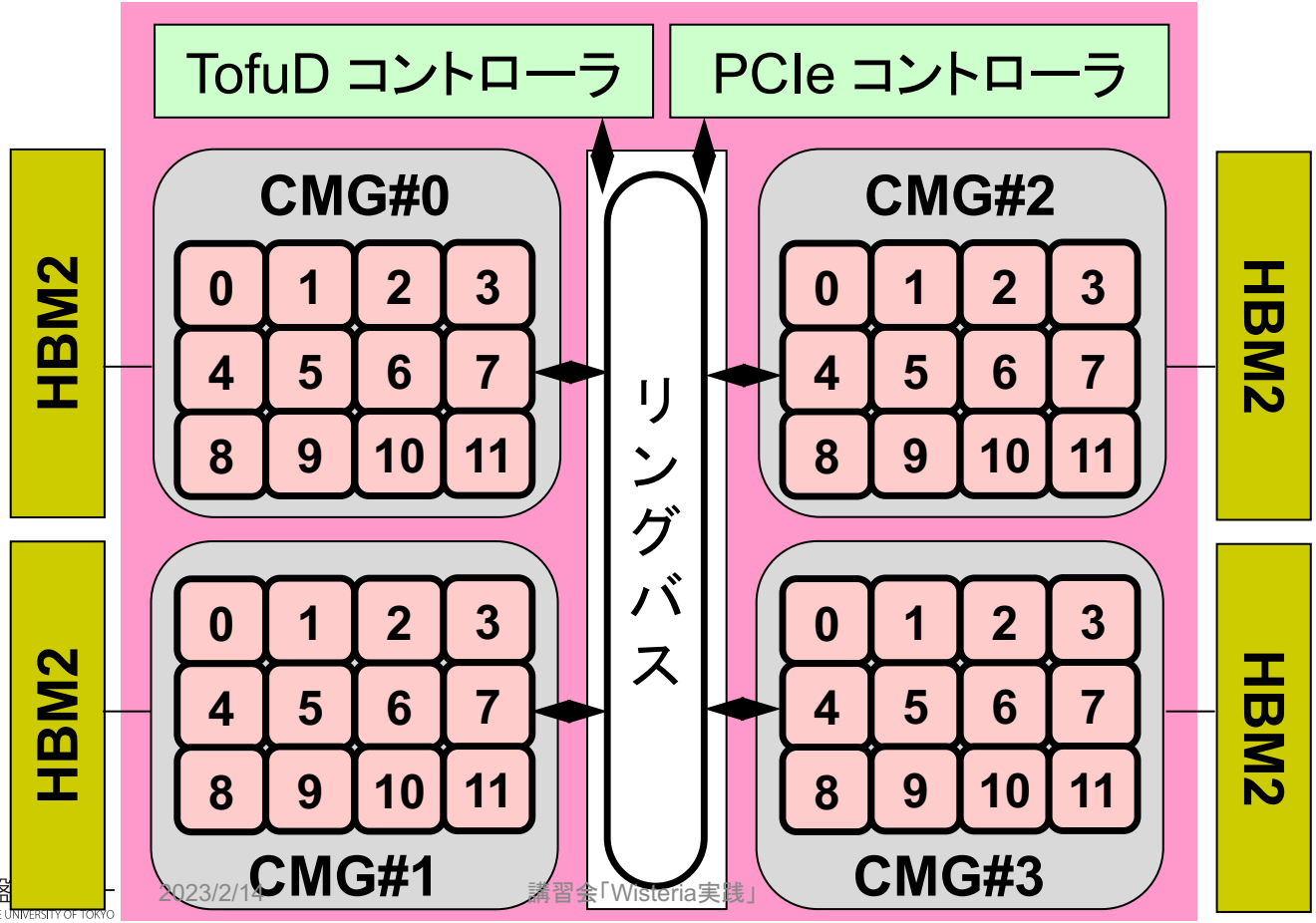
A64FXプロセッサ



プロセッサ名	Fujitsu A64FX
命令セット	Arm v8.2 +SVE (Scalable Vector Extension)
プロセッサ数 (コア数)	1 (48+アシスタントコア2 or 4)
周波数	2.2 GHz
理論演算性能	3.3792 TFLOPS
メモリ容量	32 GB
メモリ帯域幅	1,024 GB/s

- 4つのCMG（Core Memory Group），12計算コア/CMG
- NUMAアーキテクチャ（Non-Uniform Memory Access）
  - ✓ メモリは各CMGに搭載されていて独立，異なるCMGのローカルメモリ上のデータをアクセスすることは可能
  - ✓ ローカルメモリ上のデータを使って計算するのが効率的
- 大規模並列：各CMGに1-MPIプロセス（12-OpenMPスレッド），プロセッサ内4プロセスのハイブリッド推奨

A64FX：CMG（Core Memory Group）



NUMAドメイン (1/3)

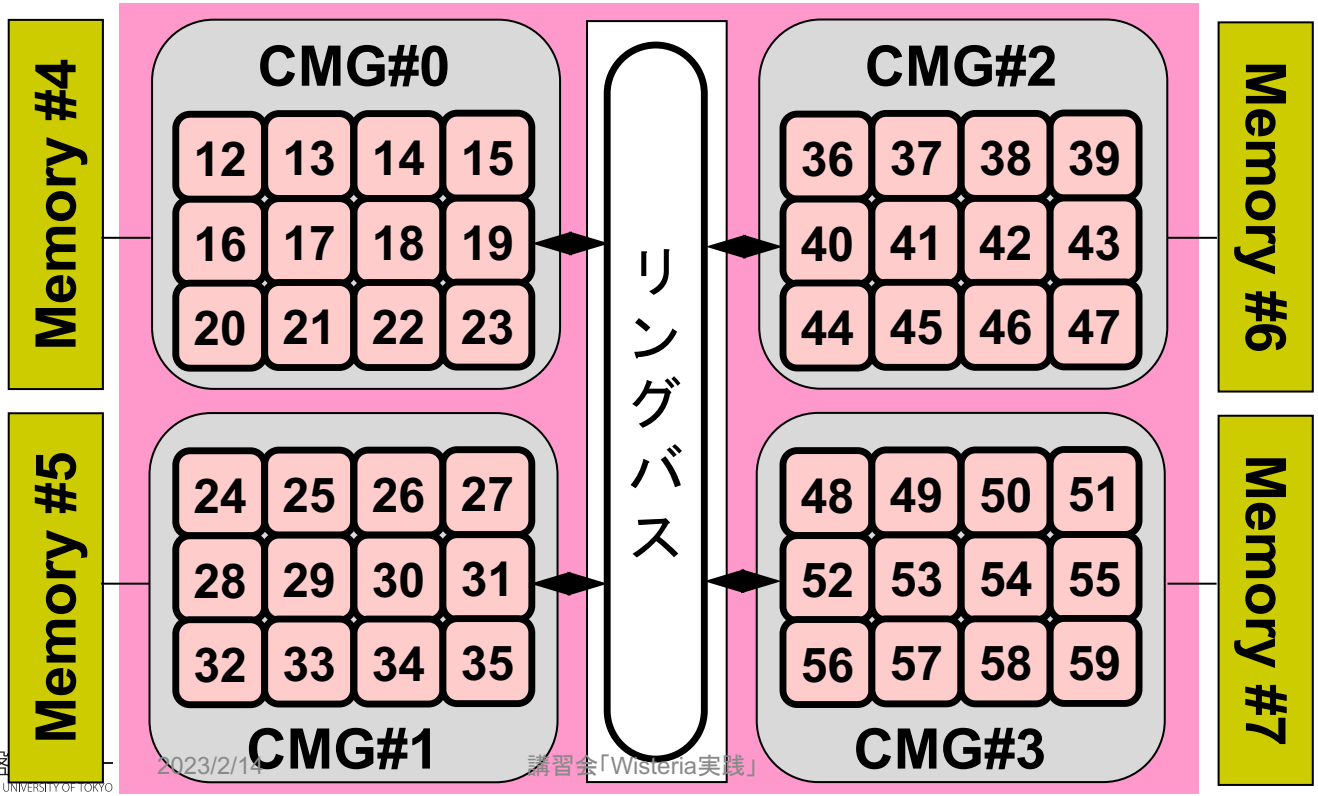
```
$ numactl -H
available: 8 nodes (0-7)
node 0 cpus:
node 0 size: 723 MB
node 0 free: 704 MB
node 1 cpus: 0
node 1 size: 765 MB
node 1 free: 394 MB
node 2 cpus: 1
node 2 size: 765 MB
node 2 free: 62 MB
node 3 cpus:
node 3 size: 759 MB
node 3 free: 678 MB
```

アシスタントコア
（OS専用、
ユーザは使えない）

```
node 4 cpus: 12 13 14 15 16 17 18 19 20 21 22 23
node 4 size: 7345 MB
node 4 free: 7205 MB
node 5 cpus: 24 25 26 27 28 29 30 31 32 33 34 35
node 5 size: 7409 MB
node 5 free: 7320 MB
node 6 cpus: 36 37 38 39 40 41 42 43 44 45 46 47
node 6 size: 7409 MB
node 6 free: 7320 MB
node 7 cpus: 48 49 50 51 52 53 54 55 56 57 58 59
node 7 size: 7402 MB
node 7 free: 7302 MB
```

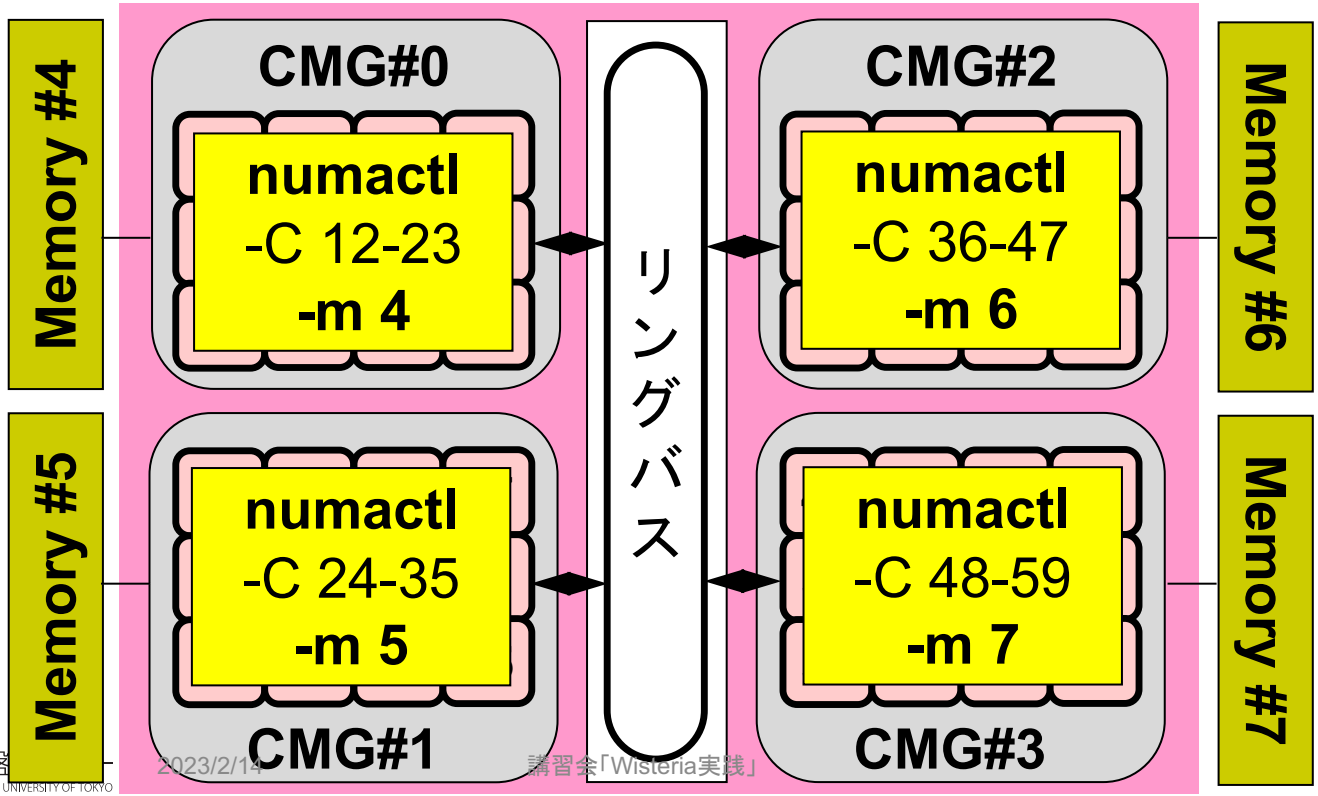
NUMAドメイン（2/3）

- **CMG:#0-#3，Core:#12-59，Memory:#4-#7**



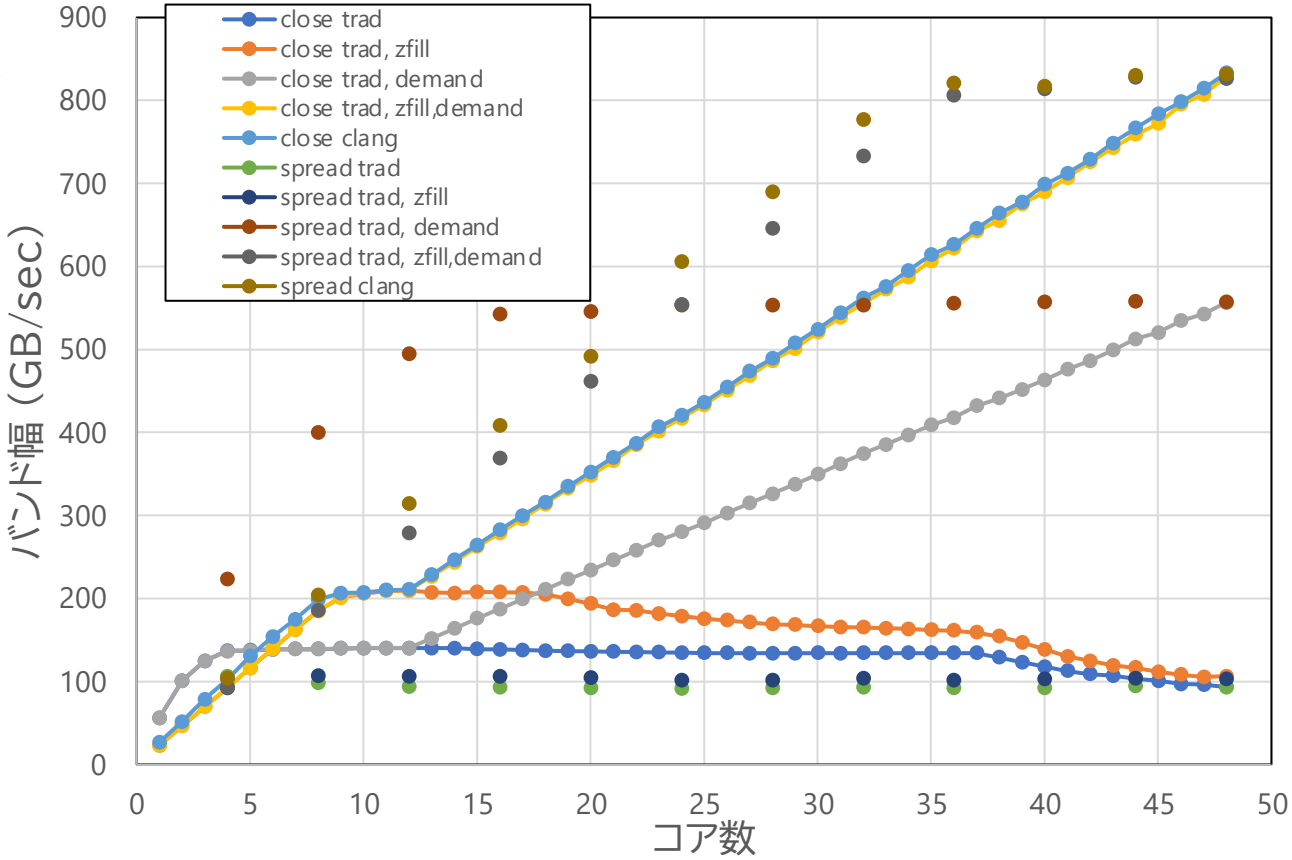
NUMAドメイン（3/3）

- **CMG:#0-#3，Core:#12-59，Memory:#4-#7**



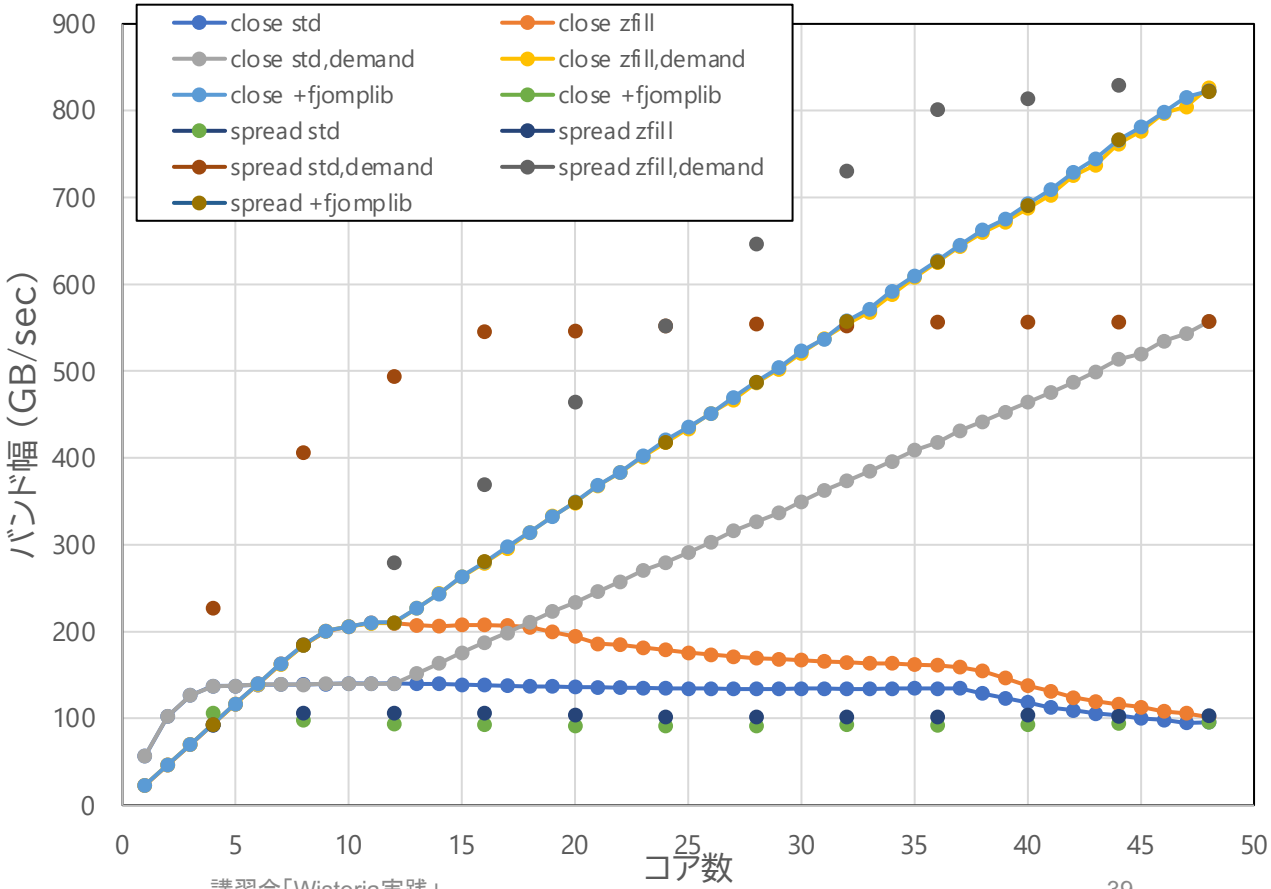
Stream Copy性能 (Odyssey, 1.2.36, C)

- close: 同一CMG優先
- spread: CMG分散
- trad: tradモード
- zfill: zfillオプション
- demand: ページ割り当て設定(後述)
- clang: clangモード(tradのオプション＋α（後述))



Stream Copy性能 (Odyssey, 1.2.36, Fortran)

- close: 同一CMG優先
- spread: CMG分散
- trad: tradモード
- zfill: zfillオプション
- demand: ページ割り当て設定(後述)
- C言語とほぼ同じ傾向



Stream Copy性能まとめ (Odyssey, 1.2.36)

- 性能が最も高い組み合わせ (~820 GB)
  - コンパイル(C, Fortran)：`-Kfast,openmp,zfill`
    実行時：`export XOS_MMM_L_PAGING_POLICY=demand:demand:demand`
  - コンパイル(C:Clangモード)：
    `-Kfast,openmp,zfill -Nclang -msve-vector-bits=512 -fno-builtin`
    実行時：`export XOS_MMM_L_PAGING_POLICY=demand:demand:demand`
- `XOS_MMM_L_PAGING_POLICY=demand:demand:demand`
  が無いと、コンパイルオプションを上の通りにしても ~ 200 GB/s
- `zfill`がないと ~560GB/s
  - `zfill`: 書き込みのみの配列をメモリを参照せずキャッシュに確保
  - 通常のコードではプリフェッチが効かず逆効果になることも多いので注意
- `libomp`(デフォルト)の場合、`spread`の方が高い性能
  - `fjomplib`の場合、`spread`配置を実現するにはコア番号を明示する必要

コンパイラの種類と実行(Odyssey)

- ログインノードとOdysseyの計算ノードとで、CPUの命令セットが大きく異なる
  - ログインノード：命令セットアーキテクチャ Intel CascadeLake＋AVX512, **x86_64**
  - Odyssey計算ノード： Fujitsu A64FX, 命令セットアーキテクチャ ARM v8.2 + SVE, **aarch64**

- 富士通製コンパイラ：`module load odyssey`
  - ネイティブコンパイラ： ＜コンパイラの種類名＞
  - クロスコンパイラ： ＜コンパイラの種類名＞+`px`
  - MPI: `mpi`+コンパイラ名 (例：`mpifccpx`)

言語	ネイティブ コンパイラ	クロス コンパイラ
C	fcc	fccpx
C++	FCC	FCCpx
Fortran	frt	frtpx

コンパイラ内部動作の切り替え (Odyssey)

- OpenMPランタイムライブラリ選択
  - -Nlibomp : デフォルト (LLVM由来)
  - -Nfjomplib : 富士通OpenMPライブラリ
- libompのとき
  - デフォルトはソフトウェアバリア、ハードウェアバリアを試すのが良い（ただしCMG内のみ有効、バグもあるかも）

```
export FLIB_BARRIER=HARD
```

- 詳細は
  - 「プログラミングガイド プログラミング共通編」
    - p.50-57 2つのOpenMPライブラリ
    - p.57 OpenMP microbenchmark
      - fjomplibの方が最大5倍速いが、OpenMP3.1までしか対応しない = task文はlibompを使う

- モード (C言語のみ)
  - -Nnoclang : tradモード (デフォルト)
  - -Nclang : clangモード(LLVMベース)
    - clangモードの方が性能改善が顕著であるので、試す価値がある
    - バグも多いかもしれない
    - libompしか使えない

可能な組み合わせ	C / C++		Fortran
	tradモード	clangモード	
libomp	○ (default)	○ (default)	○ (default)
fjomplib	○	×	○

基本的なコンパイルオプション (Odyssey)

- `-Kfast,openmp`
  - 高速なプログラムを生成
  - -O3 + いろいろ
  - 後者はOpenMPを使う場合に指定
- `-Kzfill[=N]`
  - キャッシュに書き込み領域を直接確保
    - 明らかにメモリバンド幅ネックな場合に有効
    - 本来は以下でループ毎に明示すべき

      C: `#pragma loop [no]zfill [N]`

      Fortran: `!ocl loop [no]zfill [N]`

      `-Kocl`オプションが必要
- `-Kocl`
  - 富士通の拡張機能を使う場合
- `-Koptmsg=2`
  - 最適化レポート
  - いつもつけておくことをお勧めします
- 詳細はWebで…

  ポータル➔ドキュメント閲覧
  - C言語使用手引書
  - C++言語使用手引書
  - Fortran使用手引書

基本的なコンパイルオプション (Odyssey, C, -Nclang)

- -msve-vector-bits=512
  - SVEベクトル長を512bitに固定
  - A64FXではこれが最適値
    (デフォルトではscalable)
- -ffj-zfill[=N]
  - キャッシュに書き込み領域を直接確保
    - 明らかにメモリバンド幅ネックな場合に有効
    - 本来は以下でループ毎に明示すべき
      `#pragma fj loop [no]zfill [N]`
    - -fno-builtin併用が必要な場合もある
- -ffj-ocl
  - 富士通の拡張命令を使う場合

- 詳細はWebで…
  ポータル➔ドキュメント閲覧
  - C言語使用手引書
  - C++言語使用手引書

First touch

- NUMAにおけるFirst touch
  - 初めてデータにアクセスする際に、コアが属するドメインローカルのメモリにデータが配置される
    ➡初期化等の際にも、実際の計算と同じアクセスパターンで実行する必要あり

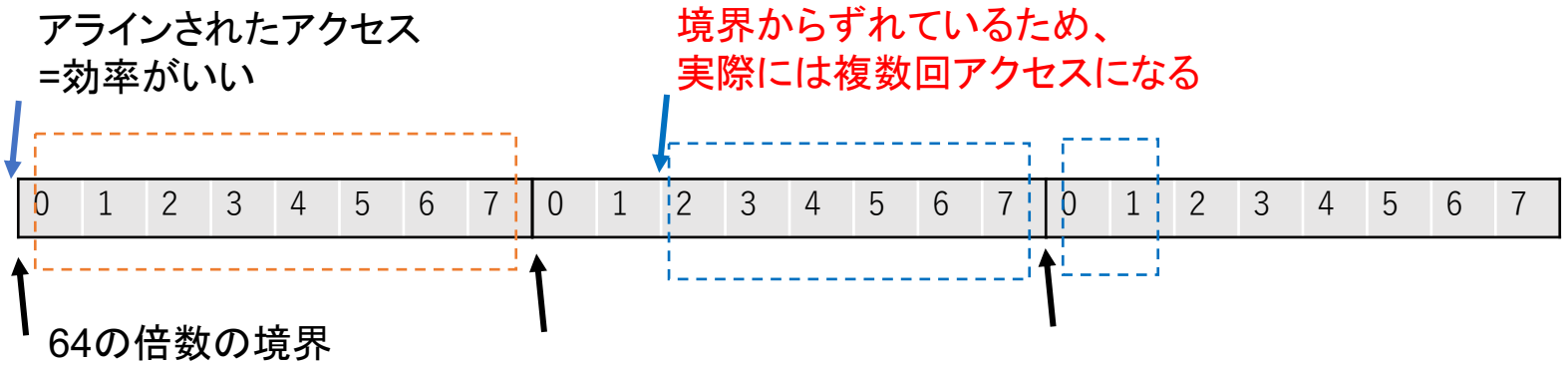
```
C(:)=0 ←全てコア0に配置されてしまう
!$omp parallel do
do i=1,100
   C(i) = C(i)+A(i)
enddo
```

```
!$omp parallel do
do i=1, 100
  C(i)=0
enddo
!$omp parallel do
do i=1, 100
   C(i) = C(i)+A(i)
enddo
```

- A64FXでもCMGを超える際の影響は大きい
  - numactlで正しいドメイン(コアとメモリの組）を指定しておく必要がある
- 実行時に必要: export XOS_MMM_L_PAGING_POLICY=demand:demand:demand

メモリのアラインメント

- SVE命令で512ビットデータを有効に使うためには、データがメモリ境界に揃っていることが必要
  - 512ビット = 64バイトの倍数のアドレスにアクセスするように
- 境界以外では複数回のアクセスになる
  - 例：倍精度実数(double)を連続8個(=512ビット)読み出す場合



メモリアラインメントの指定

- C言語
  - `__attribute__((aligned(64)))` を指定 (64 byte = 512 bit)
    - 例： `double A[1024] __attribute__((aligned(64)))`
  - 動的確保の際は `posix_memalign()` を使用
    - 上と同様の例：

```
#include <stdlib.h>
double *A;
posix_memalign(&A, 64, 8192);
```

- Fortran
  - 実行時に FLIB_ALLOC_ALIGN 環境変数を指定

    `export FLIB_ALLOC_ALIGN=64`
    - Allocate文に適用される

ポインタ指定の制限緩和 (restrict, C言語)

- ポインタを経由すると最適化の制限がかかる
  - アドレスがsourceとdestinationで重複した場合，ベクトル化するとおかしくなる(aliasing)
- restrict修飾子でaliasingはないことを宣言

```
void mycopy (double * restrict a, double * restrict b, int n)
{
   for(int i=0; i<n; i++)
       b[i] = a[i];
}
```

- またはコンパイル時にtradモードでは -Krestp を指定
  - clangモードではrestrictを指定する以外方法がない

環境変数

- ページ割り当て方式：NUMAをうまく使うには、以下の設定が必要

  ```
  export XOS_MMM_L_PAGING_POLICY=demand:demand:demand
  ```

  - ラージページかつデマンドページング（3つのメモリセクションともに）
- 他にもメモリ関連、OpenMP関連などで多様なオプションがある

- 参考：
  - ジョブ運用ソフトウェア エンドユーザ向けガイド HPC拡張機能編

まとめ：逐次実行レベル

1. CMGをまたぐ処理は性能が低下する場合が多い。MPIの併用を検討しましょう。
2. ラージページ＋デマンドページングのオプションをまずは入れましょう。
   ```
   export XOS_MMM_L_PAGING_POLICY=demand:demand:demand
   ```
3. First Touchは必須。これも2.の設定がないと機能しません。
4. メモリのアラインメントをなるべく64 byteに揃える。

最適化の弊害(?)

- あるパターンを見つけるとライブラリ関数への置き換えを勝手にやってしまう

```
for(i = 0; i < N; i++)
  B[i] = A[i];
```

→ memcpy

Clangモードの場合に起こる
(zfillが効かなくなる)
➔ -fno-builtinで抑制

```
#pragma omp parallel for private(j,k)
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    for(k=0; k<N; k++)
      C[i][j]+=A[i][k]*B[k][j]
```

→ matmul

- BLASが呼ばれるが少しでも形が崩れると適用されない
  - スカラ変数に代入したりとか
- 講習会・講義で演習をしていて何も工夫しない方が早い、という現象が起きる、、
- -Kopenmpつけずにコンパイルしても、BLASがOpenMP並列化されているため、OMP_NUM_THREADSに連動

コンパイルオプション

- `-Kfast,openmp,restp=all -x0 -Koptmsg=2`
  - `-Krestp=all`: 全てにrestrictを仮定する ➔ これがないとそもそも最適化が効かない
  - `-x0`: インライン展開を無効にする ➔ こうしないとmatmulパターンを見つけてくれない
- Optimization messages

```
jwd6002s-i  "mat-mat-openmp.c", line 35: このループをSIMD化しました。
~~jwd8204o-i  "mat-mat-openmp.c", line 35: ループにソフトウェアパイプライニングを適用しました。~~
~~jwd8205o-i  "mat-mat-openmp.c", line 35: ループの繰返し数が48回以上の時、ソフトウェアパイプライニングを適用したループが実行時に選択されます。~~
      jwd8663o-i  "mat-mat-openmp.c", line 35: ソフトウェアパイプライニングの効果がないループと判断したため、ソフトウェアパイプライニングを抑止しました。
      jwd8202o-i  "mat-mat-openmp.c", line 35: このループを展開数2回でループアンローリングしました。
jwd8330o-i  "mat-mat-openmp.c", line 36: ループ制御変数'i',...,'j'の多重ループを1重化しました。
~~jwd8101o-i  "mat-mat-openmp.c", line 57: 利用者定義の関数'MyMatMat'をインライン展開しました。~~
jwd8209o-i  "mat-mat-openmp.c", line 67: 多項式の演算順序を変更しました。
jwd6131s-i  "mat-mat-openmp.c", line 74: ループ出口が2箇所以上あるため、このループはSIMD化できません。
jwd8671o-i  "mat-mat-openmp.c", line 74: ループの形状が最適化の対象外であるため、ソフトウェアパイプライニングを適用できません。
~~jwd6004s-i  "mat-mat-openmp.c", line 95: リダクション演算を含むループ制御変数'k'のループをSIMD化しました。~~
~~jwd8204o-i  "mat-mat-openmp.c", line 95: ループにソフトウェアパイプライニングを適用しました。~~
~~jwd8205o-i  "mat-mat-openmp.c", line 95: ループの繰返し数が256回以上の時、ソフトウェアパイプライニングを適用したループが実行時に選択されます。~~
~~jwd8663o-i  "mat-mat-openmp.c", line 95: ソフトウェアパイプライニングの効果がないループと判断したため、ソフトウェアパイプライニングを抑止しました。~~
~~jwd6208s-i  "mat-mat-openmp.c", line 96: 変数'C'を定義・参照する順序がわからないため、定義・参照する順序が逐次実行と変わる可能性があり、このループはSIMD化できません。~~
~~jwd8208o-i  "mat-mat-openmp.c", line 96: ループ内の総和または乗積演算の計算方法を変更しました。~~

      jwd8220o-i  "mat-mat-openmp.c", line 93: 副作用の可能性のある最適化を行いました。
      jwd8331o-i  "mat-mat-openmp.c", line 93: このループをライブラリ呼出し(matmul)に変換しました。
```

OpenMPによるSIMD化

OpenMP+SIMD

OpenMPの基本的な内容については、以下の講習会「OpenMPによるマルチコア・メニィコア並列プログラミング入門」参照
https://www.cc.u-tokyo.ac.jp/events/lectures/154/

- SIMD (Single Instruction - Multiple Data)
  - 1命令で複数データを処理
  - SVE命令：1命令で倍精度浮動小数点数8個を処理
  - A64FXではSVE命令を最大限活用しなければ高い性能が得られない
- SIMD節はOpenMP 4.0からサポート
  - コンパイラが自動ベクトル化もしてくれるが，指示をした方が確実
  - コンパイラが誤った答えを出すこともあるので要確認（経験済み）
- コードの書き換えを必要とする場合も（しばしば）ある
- メモリのアラインメントが重要（前述）

OMP For/Do+SIMD

- C言語：

```
#pragma omp parallel for simd
for (i=0; i < num; i++) {
   sum[i] = sum[i] + a[i];
}
```

- Fortran:

```
!$omp parallel do simd
do i=1, num
   sum(i) = sum(i) + a(i)
!$omp end parallel
```

倍精度: simdlen = 8
単精度: simdlen = 16
num=>各ループが上の倍数になるように割り当て

OMP SIMD単独

- C言語：

```
#pragma omp parallel for
for (i=0; i < num; i++) {
#pragma omp simd
  for (j=0; j < num; j++) {
    sum[i] = sum[i] +a[i][j];
  }
}
```

- Fortran:

```
!$omp parallel do
do i=1, num
!$omp simd
  do j=1, num
    sum(i) = sum(i) + a(j,i)
  enddo
!$omp end simd
enddo
!$omp end parallel
```

Declare SIMD：富士通コンパイラは苦手

- 関数をまるごとSIMD化したい場合
  - その関数はOMP SIMDの中で呼ばれる場合にSIMD化される
  - インライン展開もあまり、、

```
#pragma omp declare simd notinbranch
float min (float a, float b)
{ return a < b ? a : b; }

void minner (float *a, float *b, float *c) {
#pragma omp parallel for simd
for (i=0; i<N; i++)
  c[i] = min(a[i], b[i]);
}
```

最適化の例: 「バリアフリー」

```
#pragma omp parallel for
for (i=0; i<N; i++)
  c[i] = a[i] + b[i];
/* 暗黙のバリア同期 */
/* 一度single regionに戻る*/
#pragma omp parallel for
for (i=0; i<N; i++)
  f[i] = d[i] + e[i];
```

```
#pragma omp parallel
{
#pragma omp for nowait
 for (i=0; i<N; i++)
   c[i] = a[i] + b[i];
#pragma omp for
 for (i=0; i<N; i++)
   f[i] = d[i] + e[i];
} // parallel終わり
```

forの終了時にバリア同期しない

まとめ：OpenMPレベル

- C言語：OpenMPをちゃんと使うなら、tradコンパイラよりclangコンパイラの方がよい
  - 今後の開発はclangベースで行われる

Wisteria-Odyssey
MPI+OpenMPハイブリッド

MPIに関する情報

- 「MPI基礎：並列プログラミング入門」講習会資料
  - https://www.cc.u-tokyo.ac.jp/events/lectures/189/
- 「MPI上級編」講習会資料
  - https://www.cc.u-tokyo.ac.jp/events/lectures/190/

- MPI仕様書 (3.1)
  - https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
- MPI仕様書 (4.0)
  - https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

Wisteria-OdysseyのTofu-D

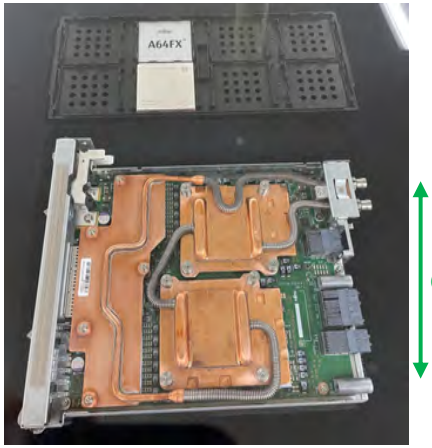
- Tofu-D 形状: 全20ラック
  - (X, Y, Z, a, b, c) = (10, 8, 8, 2, 3, 2)

➔ 3次元、2次元、1次元に展開して指定可能

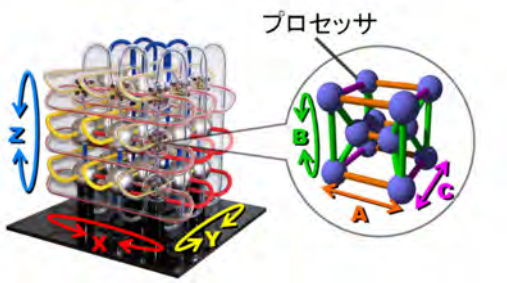
典型的な例（全系）

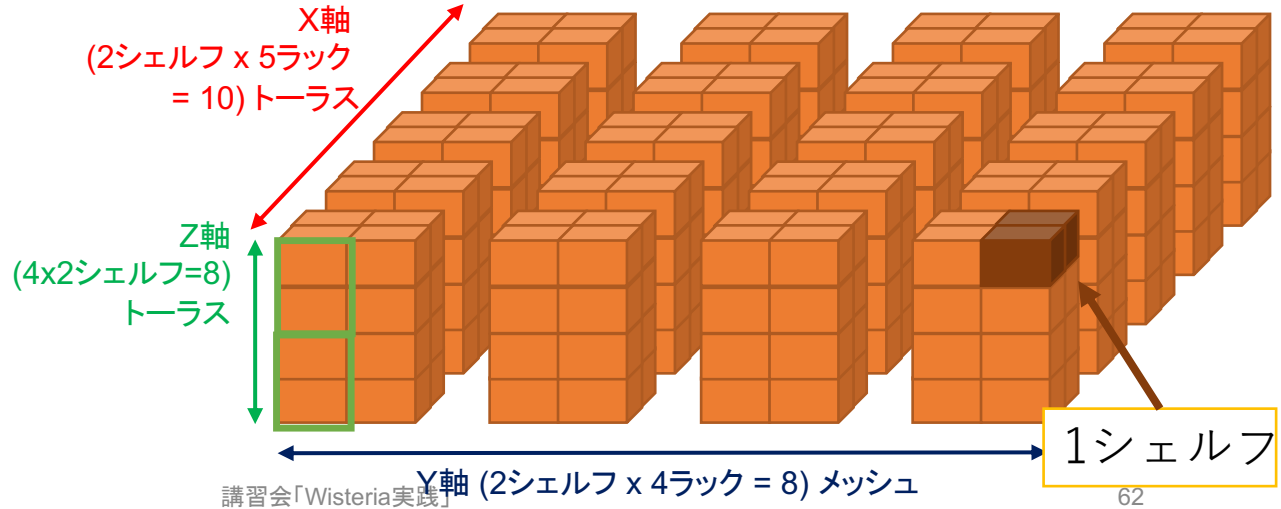
- 3次元: 20 x 24 x 16
  - X*a, Y*b, Z*c
- 2次元: 60 x 128
  - X*a*b, Y*Z*c
- 1次元: 7680

シェルフ内に
24枚搭載
シェルフ当たりは
(1, 1, 4, 2, 3, 2)

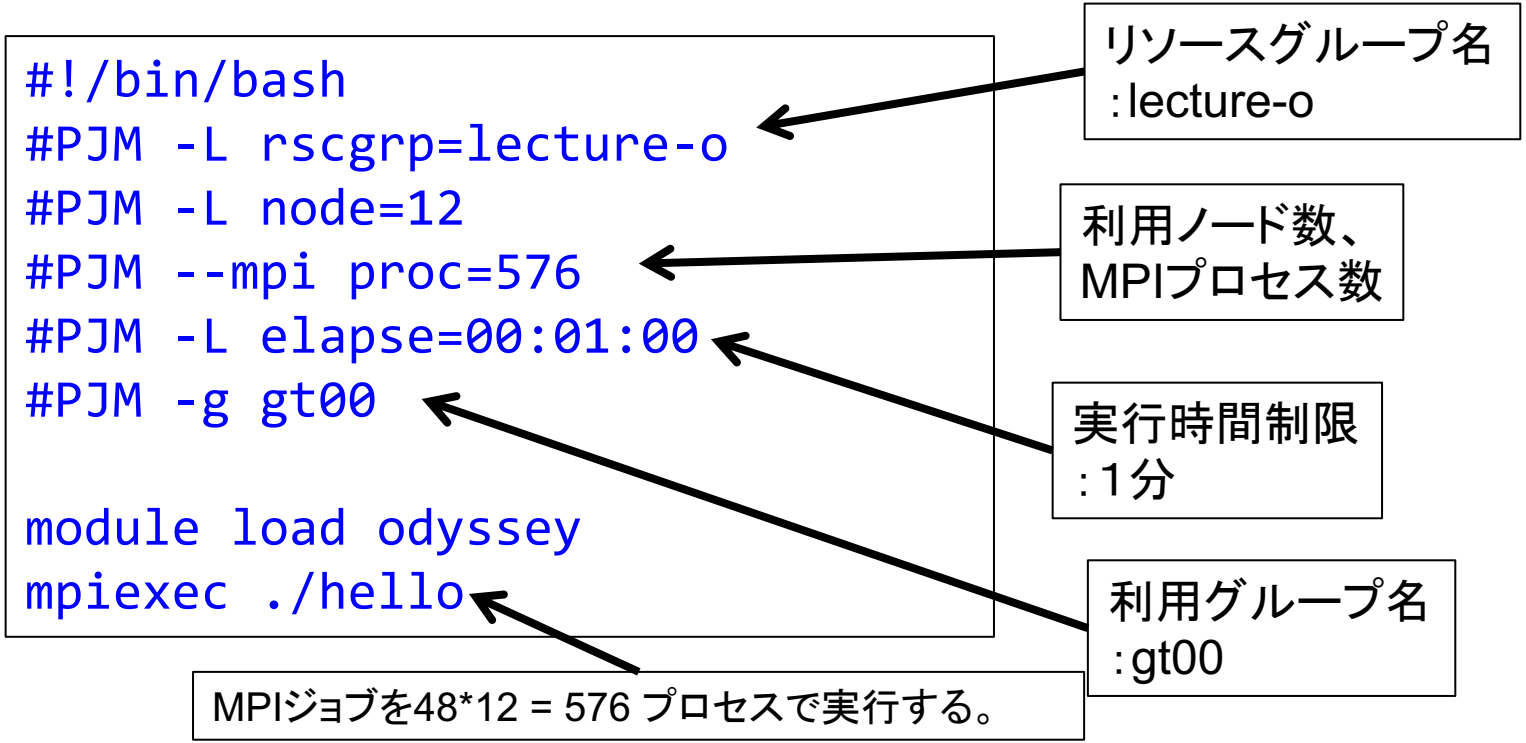


参考：
ジョブ運用ソフトウェア
エンドユーザ向けガイド
MPI使用手引書





JOBスクリプト（Odyssey, フラットMPI）



JOBスクリプト（Odyssey, ハイブリッドMPI: 1プロセス/ノード）

```
#!/bin/bash
#PJM -L rscgrp=lecture-o
#PJM -L node=12
#PJM --mpi proc=12
#PJM --omp thread=48
#PJM -L elapse=00:01:00
#PJM -g gt00

module load odyssey
mpiexec ./hello_omp
```

リソースグループ名
：lecture-o

利用ノード数

利用コア数
（MPIプロセス数）

1MPIプロセス当たり
48スレッド生成

実行時間制限：1分

利用グループ名
：gt00

MPIジョブを1＊12 = 12 プロセスで実行する。

JOBスクリプト（Odyssey, ハイブリッドMPI: 4プロセス/ノード）

```
#!/bin/bash
#PJM -L rscgrp=lecture-o
#PJM -L node=12
#PJM --mpi proc=48
#PJM --omp thread=12
#PJM -L elapse=00:01:00
#PJM -g gt00

module load odyssey
mpiexec ./hello_omp
```

リソースグループ名
:lecture-o

利用ノード数

利用コア数
（MPIプロセス数）

1MPIプロセス当たり
12スレッド生成

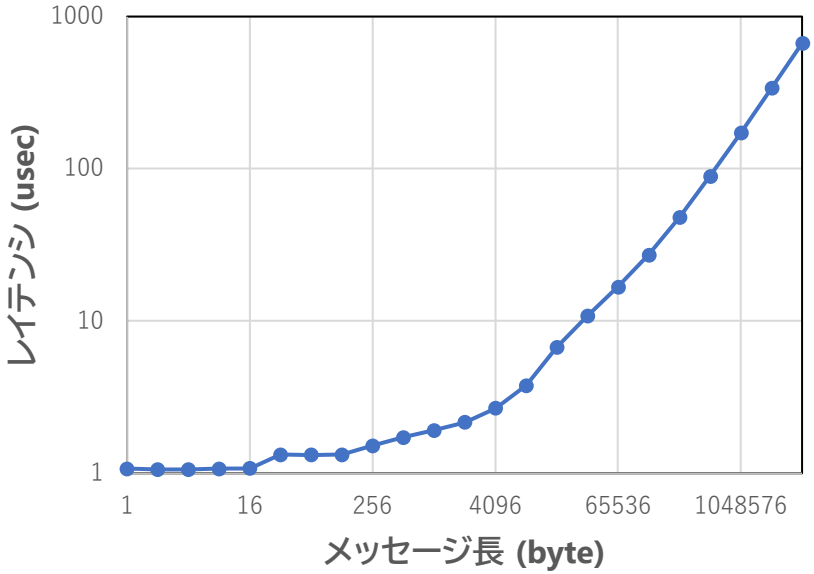
実行時間制限：1分

利用グループ名
:gt00

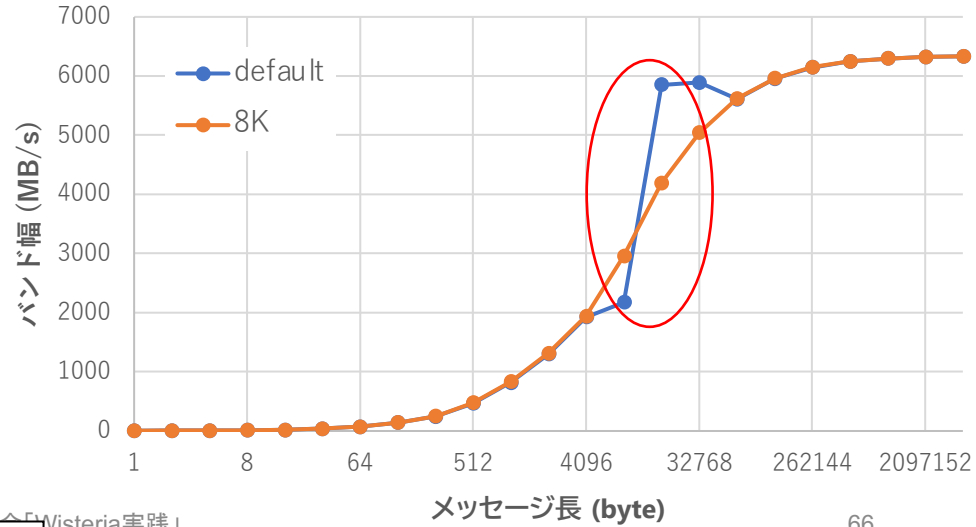
MPIジョブを4＊12 = 48 プロセスで実行する。

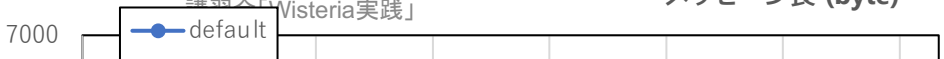
MPI性能 (Odyssey, pt2pt, 1.2.37)

- レイテンシ：osu_latency
  - 最小(隣接): **1.06us**



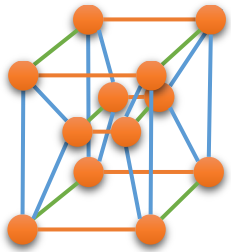
- バンド幅: osu_bw
  - 最大(隣接): **6.3 GB/s**
  - 8KBの落ち込み:
    `-mca btl_tofu_eager_limit 8192`で改善も
    16K-32Kは悪化





ノード形状の指定とrank-map

- 形状の指定（-L node）
  - node=N_1
  - node=N_1xN_2
  - node=N_1xN_2xN_3
- 割り当て方の指定
  - node=N_1xN_2xN_3:torus
    - TOFU単位（12ノード: 2x3x2単位）で切り出す
  - node=N_1xN_2xN_3:mesh
    - 隣接していればいい
  - node=N_1xN_2xN_3:nocont
    - 離れていても良い
    - 性能は低下してもいいからとにかく実行したい場合



- MPIオプション（--mpi）
  - proc=プロセス数
  - rank-map-bychip,rank-map-hostfile=ホストファイル
    - ホストにN個ずつ割り当て（proc/node）
  - rank-map-bynode,rank-map-hostfile=ホストファイル
    - ホストに1個ずつ割り当て
- node=2x3の例
  - (0,0) (0,1) (0,2) (1,0) (1,1) (1,2)
    実際には各行に1要素

性能プロファイラ (Odyssey)

- 富士通コンパイラには、性能プロファイラ機能がある
- 富士通コンパイラでコンパイル後、実行コマンドで指定し利用する
- 以下の3種類があります
- 基本プロファイラ
  - プログラム全体で、最も時間のかかっている関数を同定するなど、傾向の把握
- 詳細プロファイラ
  - 最も時間のかかっている関数内の特定部分において、メモリアクセス効率、キャッシュヒット率、スレッド実行効率、MPI通信頻度解析等の詳細分析
- CPU 性能解析レポート
  - 特定区間のCPU性能解析情報の可視化
- 詳細は、
  - 「利用手引書」
  - J2UL-2483-02Z0(10)　2022年3月, Technical Computing Suite V4.0L20、Development Studio プロファイラ使用手引書

性能最適化情報の提示（1/2）

- C, Fortranを問わず、コンパイラが行った最適化情報を知ることは性能最適化で重要である
- 各ファイルのソースコードごとに、どのような最適化が行われたか出力する、コンパイラオプションがある
- C, Fortranで共通の翻訳情報
  - -Nlst=p ：標準の最適化情報を出力(デフォルト)
  - -Nlst=t ：詳細な最適化情報を出力

性能最適化情報の提示（2/2）

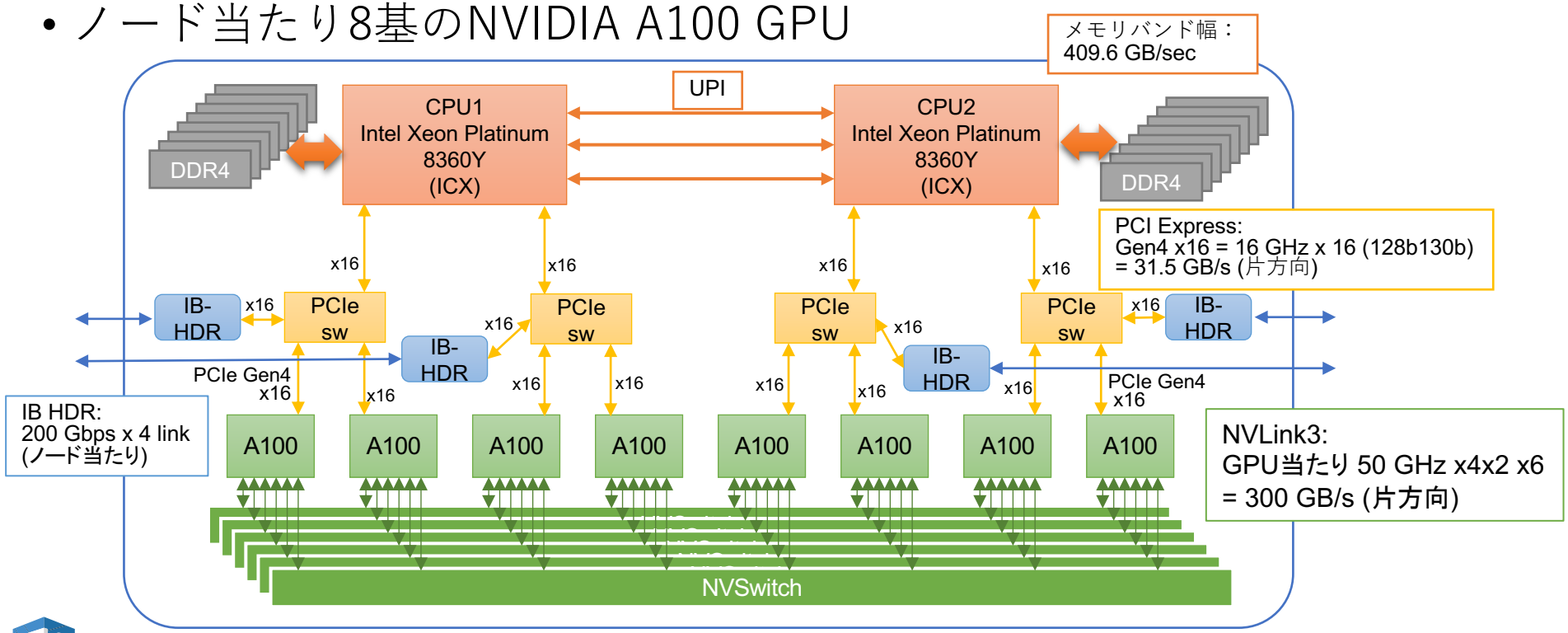
- Fortran のみ、p、t以外も指定可能
  - -Nlst=a ：名前の属性情報を出力
  - -Nlst=d ：派生型の構成情報を出力
  - -Nlst=I　：インクルードされたファイルのプログラムリスト
    およびインクルードファイル名一覧を出力を出力
  - -Nlst=m ：自動並列化の状況をOpenMP指示文によって
    表現した原始プログラムを出力
  - -Nlst=x ：名前および文番号の相互参照情報を出力
  - 詳細は、オンラインマニュアルを参照のこと
    - C言語使用手引書
    - C++言語使用手引書
    - Fortran使用手引書

Wisteria-Aquarius

GPU利用を中心に

Aquariusの構成

- Intel Xeon Platinum 8360Y (36c 2.4GHz) x 2ソケット, 512GBメモリ
- ノード当たり8基のNVIDIA A100 GPU



CPUの構成

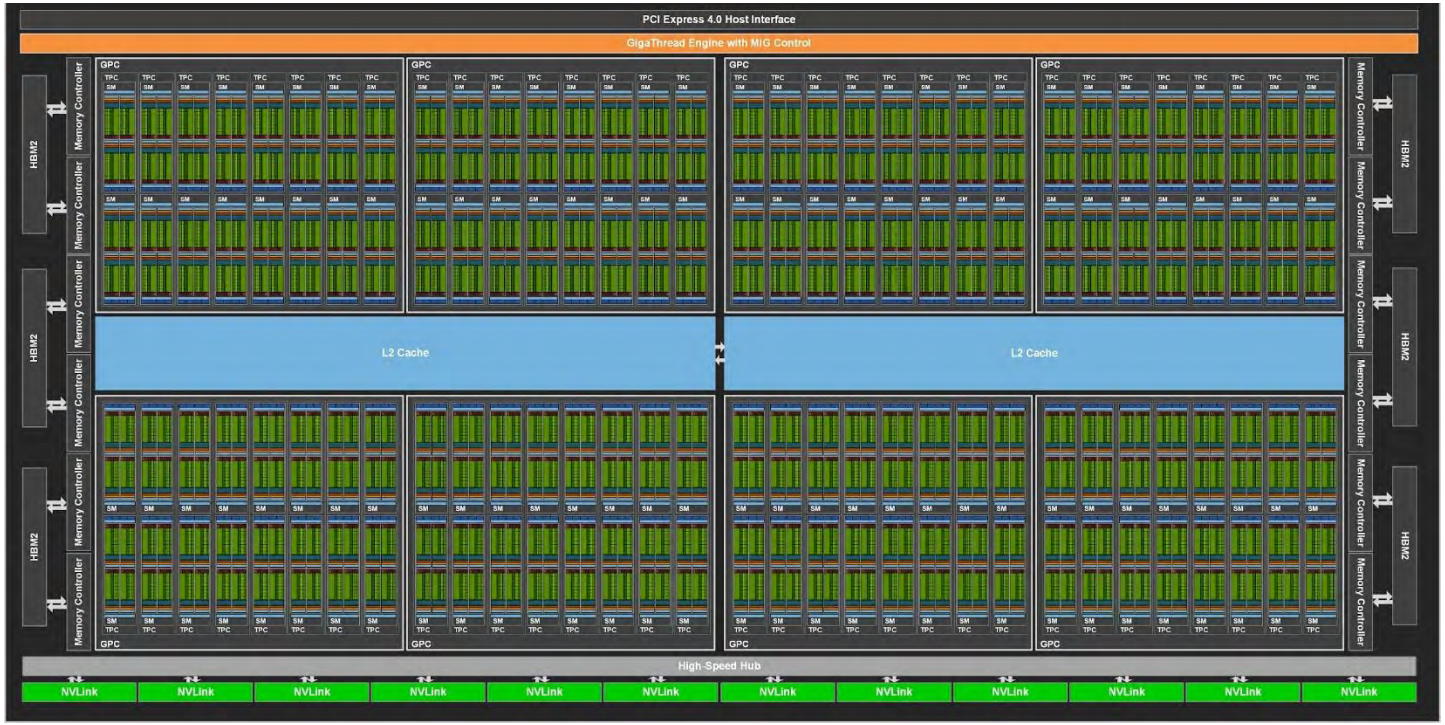
- 最新のIceLakeだが、トークン消費がGPU基準なのでCPUのみの利用はお勧めできない (ノード当たり、Odysseyの24倍の消費)
- ソケット内も2 NUMAドメインに分割 (18コアずつ)
- shareでは、1GPU当たり9コア

`nvidia-smi topo -mp`の出力 (`lecture-a`の場合)

```
      GPU0 mlx5_0 mlx5_1 mlx5_2  mlx5_3 CPU Affinity NUMA Affinity
GPU0     X   PXB    SYS    SYS    SYS   0-8         0-3
mlx5_0 PXB    X     SYS    SYS    SYS
mlx5_1 SYS   SYS     X     SYS    SYS
mlx5_2 SYS   SYS    SYS     X     SYS
mlx5_3 SYS   SYS    SYS    SYS     X
```

NVIDIA A100 Tensor Core GPU (1/2)

- 108 SM (Streaming Multiprocessor)



出典: NVIDIA A100 Tensor core GPUアーキテクチャ

NVIDIA A100 Tensor Core GPU (2/2)

- 倍精度にも対応したTensor Coreを搭載
  - 19.5 TF @ FP64, FP32
  - 156 TF @ TF32 (実質19bit)
  - 312 TF @ FP16, BF16
  - 624 TF @ INT8
  - 1248 TF @ INT4
- メモリ HBM2 40GB 搭載
  - 1.555 TB/s

出典: NVIDIA A100 Tensor core GPUアーキテクチャ

Volta世代のおさらい

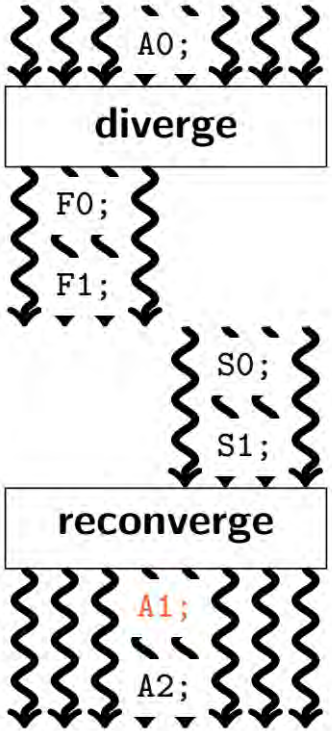
- Independent thread schedulingが導入され，（デフォルトでは）warpを構成する32スレッド内の暗黙の同期がなくなった
  - コンパイル時に arch=compute_60,code=sm_70 を指定すれば，Pascal世代以前と同様に暗黙の同期が復活
    （勝手にPascalモードと呼ぶことにしている）
  - A100ではarch=compute_60,code=sm_80で暗黙同期が復活
  - 重力ツリーコードの場合，Pascalモードの方が最大2割程度高速だった
    - 暗黙同期の有効化は最適化手法の1つ
    - A100の場合，Pascalモードは最大1割程度高速
      （差が縮まったのは，以降で紹介する新機能による高速化が効いている）
- 整数演算ユニットがCUDAコアから独立したため，独立な演算の同時実行による演算時間の隠ぺいによって高速化された

Independent thread scheduling

Pseudocode

```
A0;
if( threadIdx.x < 4 ){
  F0;
  F1;
} else {
  S0;
  S1;
}
A1;
#if __CUDA_ARCH__ >= 700
__syncwarp();
#endif
A2;
```

Pascal or earlier

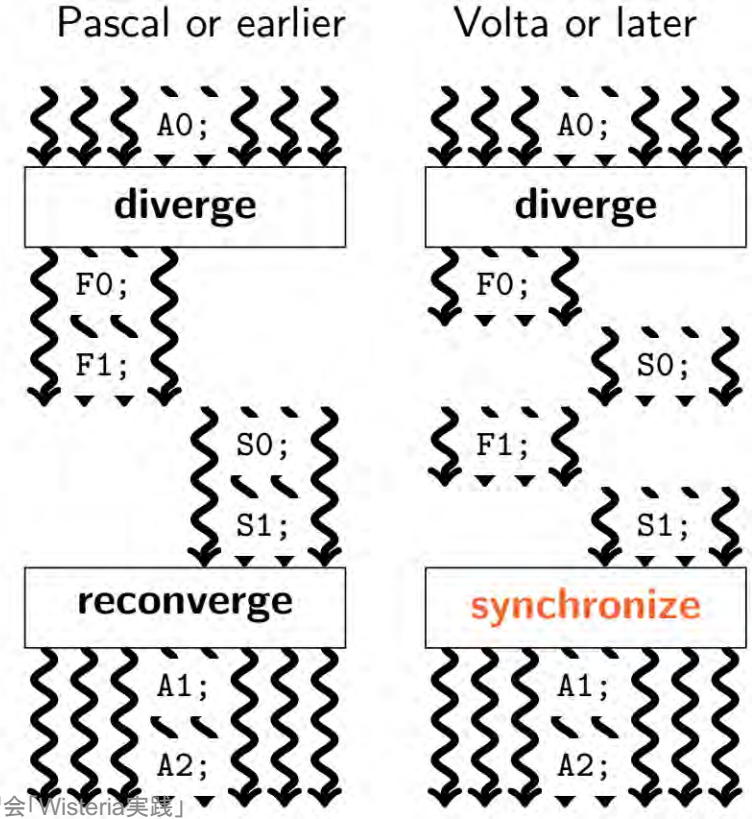


Volta or later

Independent thread scheduling

Pseudocode

```
A0;
if( threadIdx.x < 4 ){
  F0;
  F1;
} else {
  S0;
  S1;
}
#if __CUDA_ARCH__ >= 700
__syncwarp();
#endif
A1;
A2;
```



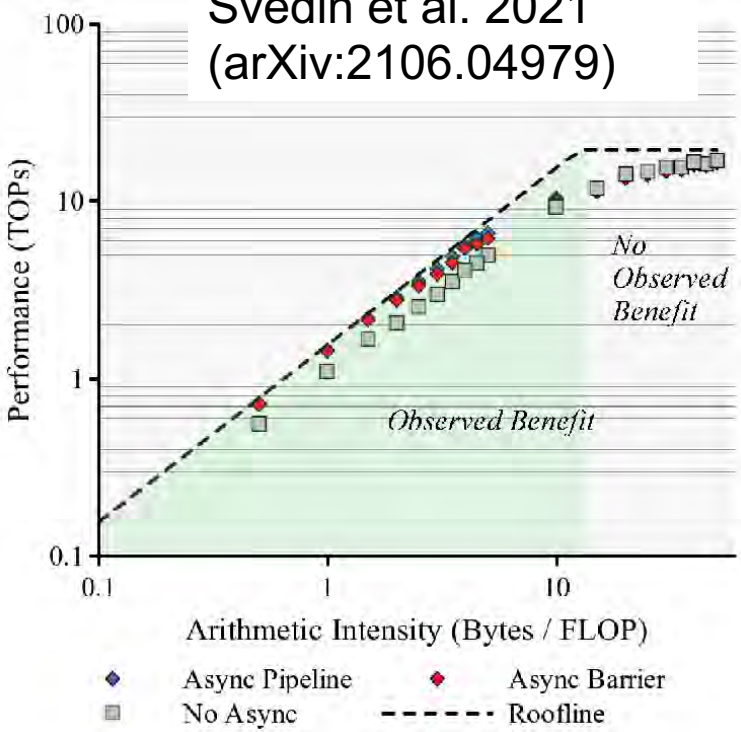
A100/CUDA 11から導入された新機能

- （正式名称NVIDIA A100 Tensor Core GPUにも入っている）テンソルコアは行列の積和算専用機であり万人向けではないため，以下では関連機能も含めてすべてスキップ
  - Sparsity, TF32（実態はFP19）がテンソルコア関連機能
  - テンソルコアなしでは，理論ピーク性能比はV100 SXM2比で1.24倍
- Asynchronous copy/barrier
  - グローバルメモリからシェアードメモリに直接（レジスタを介さずに）データを置ける
- L2 cache residency control
  - 配列要素の一部をL2キャッシュに置きっぱなしにしておける
- Warp-wide reduction
  - Warp内でのリダクション演算用の高速な命令が提供された

Asynchronous copy (`memcpy_async`)

- グローバルメモリからシェアードメモリに直接（レジスタを介さずに）データを置ける
- 処理はバックグラウンドで走る（したがって，同期も必要）
  - MPI_Send/MPI_Isend と対応させるとイメージしやすい
- A100では，hardware accelerated
  - つまり，非同期で動かせない状況でも速くなる
- Svedin et al. (2021):
  - Pipeline API は Barrier API よりも高速
  - メモリ律速な問題では性能向上
  - 演算律速な問題では性能低下

Svedin et al. 2021
(arXiv:2106.04979)



実装例（複数通り実装方法がある）

```
#include <cooperative_groups.h>
#include <cooperative_groups/memcpy_async.h>
Using namespace cooperative_groups;

// デバイス関数内での実装:
  thread_block_tile<TSUB> tile = tiled_partition<TSUB>(this_thread
_block());
  // TSUB は，この関数内で memcpy_async に関与するスレッド数（この書き方の場
合は32以下）

  cooperative_groups::memcpy_async(tile, &shared_mem, &global_mem,
 sizeof(uint) * num);
  // 何か裏で回したい処理があれば，ここに書く
  cooperative_groups::wait(tile);// ここでコピーの完了待ちをする
```

NVIDIA Developer Blogでの実装例

- https://developer.nvidia.com/blog/cuda-11-features-revealed/

```
//Without async-copy
__shared__ extern int smem[];

// algorithm loop iteration
 while ( ... ) {
  __syncthreads();
  // load element into shared mem
  for (int i = ... ) {
    smem[i] = gmem[i];
  }
  __syncthreads();

  /* compute on smem[] */
}
```

```
//With async-copy
using namespace nvcuda::experimental;
__shared__ extern int smem[];
pipeline pipe;

// algorithm loop iteration
 while ( ... ) {
  __syncthreads();
  // load element into shared mem
  for (int i = ... ) {
    // initiate async memory copy
    memcpy_async(smem[i], gmem[i], pipe
);
  }
  // wait for async-copy to complete
  pipe.commit_and_wait();
  __syncthreads();

  /* compute on smem[] */
}
```

L2 cache residency control

- Best Practice Guide中の記述：
  "A portion of the L2 cache can be set aside for persistent accesses to a data region in global memory"
- L2キャッシュの容量は40 MB（V100では6 MBだった）
  - 1/16（=2.5 MB）刻みで調整できる（white paperより）
- CUDAストリームごとに1つの配列中の連続領域を指定可能
  - 16個設定できても良さそうだが，残念ながらそうなってはいない

NVIDIA Developer Blogでの実装例

- https://developer.nvidia.com/blog/cuda-11-features-revealed/

```
cudaStreamAttrValue attr ;

attr.accessPolicyWindow.base_ptr = /* beginning of range in global memory */ ;
attr.accessPolicyWindow.num_bytes = /* number of bytes in range */ ;

// hitRatio causes the hardware to select the memory window to designate as per
sistent in the area set-aside in L2
attr.accessPolicyWindow.hitRatio = /* Hint for cache hit ratio: 1.0で良い */

attr.accessPolicyWindow.hitProp = cudaAccessPropertyPersisting;
attr.accessPolicyWindow.missProp = cudaAccessPropertyStreaming;

cudaStreamSetAttribute(stream, cudaStreamAttributeAccessPolicyWindow, &attr);
```

Warp-wide reduction

- Throughputは16（FP32演算が64なので，4演算相当）
  - Warp shuffleが32（2演算相当）なので，5段（32スレッド）必要な旧実装に比べて（演算コストを無視しても）圧倒的に速い
- `__reduce_*_sync(unsigned mask, T value)`
  - T: unsigned/intに対してはadd, min, max
    - 小細工すればfloatに対してmin/maxを返させることは可能
  - T: unsignedに対してはand, or, xor も可能
  - Supported by devices of compute capability 8.x or higher
    - Pascalモード（arch=compute_60,code=sm_80）の場合には，コンパイルが通らなかった（compute_80の指定が必要）
    - Pascalモード使用による高速化か，Ampereモード＋warp-wide reductionによる高速化か，の競争（コード次第なので，どちらが速いかはお試しください）

シェアードメモリ使用時の注意点

- 静的に確保できるのはブロック当たり48 KBまで（この制約はずっと昔からあった）
  - V100ではSMあたり96 KBまでシェアードメモリに使えたが，（常識的には）SMあたりに複数ブロック立てるため，実質的に制約なしだった
  - A100では，SMあたり164 KBまで使える（1ブロックでは163 KBまで）
  - Dynamic allocationを使えば48 KB越えでも問題ない

```
extern __shared__ real dynamic_shared_memory[];
__global__ void calc_acc(int num){
  float4* pj = (float4*)dynamic_shared_memory;
  uint* queue = (uint *)&pj[num];
  .. Skip ..
}

// how to launch
calc_acc<<<blck, thrd, size_of_shared_memory, CUDA_stream>>>(num);
```

GPU情報の表示

- nvidia-smi

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.57.02    Driver Version: 470.57.02    CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA A100-SXM...  On   | 00000000:C7:00.0 Off |                    0 |
| N/A   24C    P0    51W / 400W |      0MiB / 40536MiB |      0%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

- nvidia-smi -q

```
Timestamp                                 : Wed Sep 14 20:09:53 2022
Driver Version                            : 470.57.02
CUDA Version                              : 11.4

Attached GPUs                             : 1
GPU 00000000:C7:00.0
    Product Name                          : NVIDIA A100-SXM4-40GB
……
Applications Clocks
        Graphics                          : 1095 MHz
        Memory                            : 1215 MHz
    Default Applications Clocks
        Graphics                          : 1095 MHz
        Memory                            : 1215 MHz
    Max Clocks
        Graphics                          : 1410 MHz
        SM                                : 1410 MHz
        Memory                            : 1215 MHz
        Video                             : 1290 MHz
    Max Customer Boost Clocks
        Graphics                          : 1410 MHz
```

コンパイラの種類と実行(Aquarius)

- ログインノードとAquarius計算ノードとでは、CPUの命令セットが（ほぼ）同じ
  - ログインノード：命令セットアーキテクチャ Intel CascadeLake＋AVX512, **x86_64**
  - Aquarius計算ノード： 命令セットアーキテクチャ Intel IceLake＋AVX512, **x86_64**
- 様々なコンパイラが利用可能: GPU向けには gcc+CUDAか NVIDIAを推奨

```
$ module load aquarius cuda ompi-cuda
    (2023/2現在、gcc=8.3.1, CUDA=11.4, OMPI-CUDA=4.1.1-11.4)
$ module load nvidia cuda ompi-cuda
    (2023/2現在、 NVIDIA=22.7, CUDA=11.4, OMPI-CUDA=4.1.4-11.4)
```

言語	GNUコンパイラ	Intelコンパイラ	NVIDIA コンパイラ(旧PGI)	CUDAコンパイラ
C	gcc	icc	nvc (pgcc)	nvcc
C++	g++	icpc	nvc++(pgc++)	
Fortran	gfortran	ifort	nvfortran (pgfortran)	
OpenACC			○	

NVIDIAコンパイラ(OpenACC)

- コンパイラオプションとして
  -ta=tesla,cc80 を付ける

ソースコード

```
8.    subroutine acc_kernels()
9.      double precision :: A(N,N), B(N,N)
10.     double precision :: alpha = 1.0
11.     integer :: i, j
12.     A(:,:) = 1.0
13.     B(:,:) = 0.0
14.     !$acc kernels
15.     do j = 1, N
16.       do i = 1, N
17.         B(i,j) = alpha * A(i,j)
18.       end do
19.     end do
20.     !$acc end kernels
21.   end subroutine acc_kernels
```

サブルーチン名

コンパイラメッセージ(fortran)

```
pgfortran -O3 -acc -Minfo=accel -ta=tesla,cc80 -Mpreprocess  acc_compute.f90 -o
acc_compute
acc_kernels:
     14, Generating implicit copyin(a(:,:))
         Generating implicit copyout(b(:,:))
     15, Loop is parallelizable
     16, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         15, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
         16, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
    ....
```

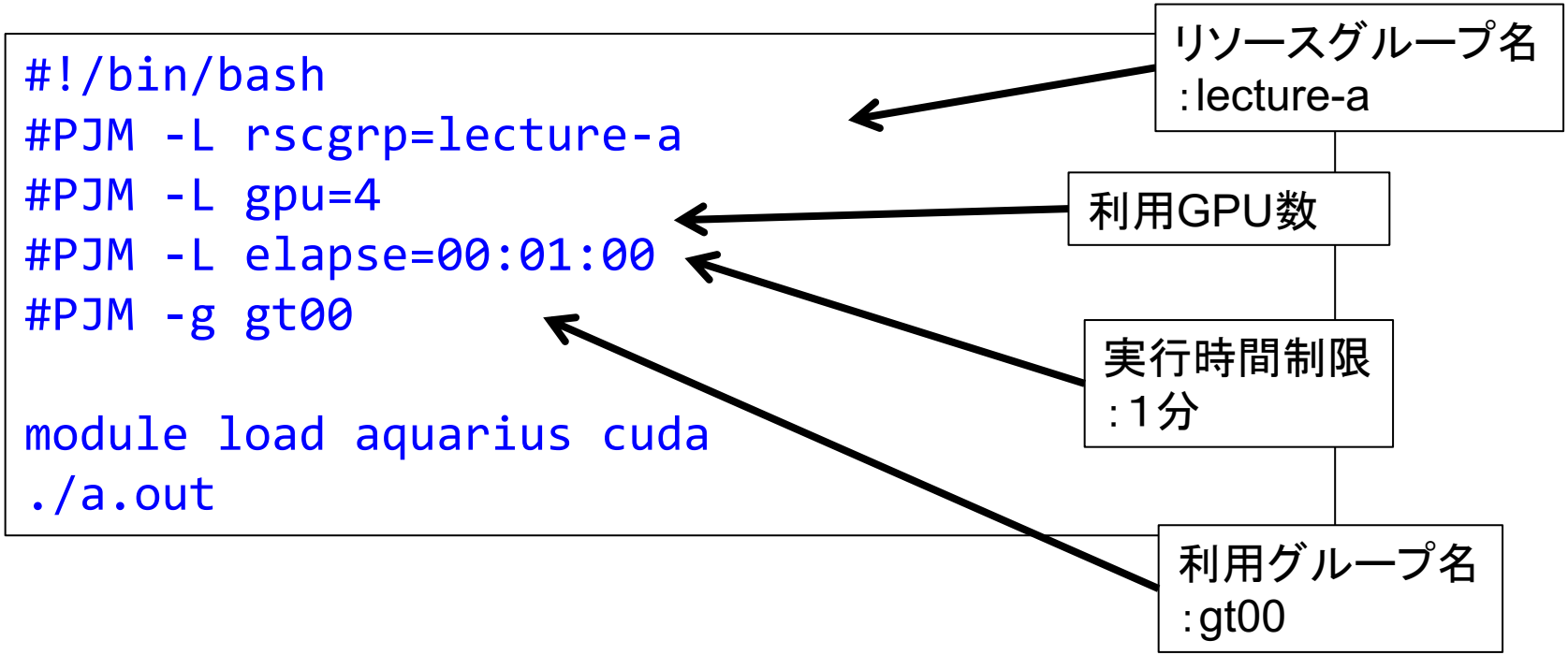
配列aはcopyin, bはcopyoutとして扱われます

15, 16行目の2重ループは(32x4)のスレッドでブロック分割して扱います。

JOBスクリプトサンプルの説明（Aquarius, MPIなし）

```
#!/bin/bash
#PJM -L rscgrp=lecture-a
#PJM -L gpu=4
#PJM -L elapse=00:01:00
#PJM -g gt00

module load aquarius cuda
./a.out
```



Wisteria-Aquarius GPU+MPI環境

OpenMPI

- Aquariusでは、MPIとGPUを使うのは OpenMPIがお勧め
  - CUDA Aware: GPUメモリを直接送受信可能(論理的に)
  - GPU Direct RDMA: さらにRDMA機能で高速に転送
  - デフォルトでこれらの機能がONになっている

- `export UCX_MAX_RNDV_RAILS=1`　（ノード内通信では増やすべき）
- `export UCX_RNDV_THRESH=32K`
  - 特定メッセージサイズで性能が落ち込むときに調整
- 不具合が起こる時：以下をお試しください
  - `export UCX_MEMTYPE_CACHE=n`
  - `export UCX_IB_GPU_DIRECT_RDMA=n`

JOBスクリプトサンプルの説明（Aquarius, MPI）

```
#!/bin/bash
#PJM -L rscgrp=lecture-a
#PJM -L gpu=4
#PJM --mpi proc=4
#PJM -L elapse=00:01:00
#PJM -g gt00

module load aquarius cuda ompi-cuda
mpiexec -machinefile $PJM_O_NODEINF -n $PJM_MPI_PROC ¥
-npernode 4 ./wrapper.sh ./a.out
```

リソースグループ名
：lecture-a

利用GPU数、
MPIプロセス数

実行時間制限：1分

利用グループ名
：gt00

MPIジョブをノードあたり4プロセスで実行する。

Aquariusノード内で複数ランク使う場合

- プログラム側で複数GPUを考慮していない場合には、以下のようなシェルスクリプトを用意
- wrapper.sh (chmod +x wrapper.sh を忘れずに)

```
#!/bin/sh
export LOCAL_RANK=$OMPI_COMM_WORLD_LOCAL_RANK
export CUDA_VISIBLE_DEVICES=$LOCAL_RANK
$*
```

Aquariusノード内で複数ランク使う場合(2)

- どのネットワークデバイスを使うか考慮
  - UCX_NET_DEVICES 環境変数
  - InfiniBandのポート番号を使い分ける
- 例えば1レールずつ分けるとwrapper.shはこうなる

```
#!/bin/sh
case $OMPI_COMM_WORLD_LOCAL_RANK in
        0|1) export UCX_NET_DEVICES=mlx5_0:1 ;;
        2|3) export UCX_NET_DEVICES=mlx5_1:1 ;;
        4|5) export UCX_NET_DEVICES=mlx5_2:1 ;;
        6|7) export UCX_NET_DEVICES=mlx5_3:1 ;;
esac
$*
```

Aquariusノード内で複数ランク使う場合(3)

- まとめると以下の通り

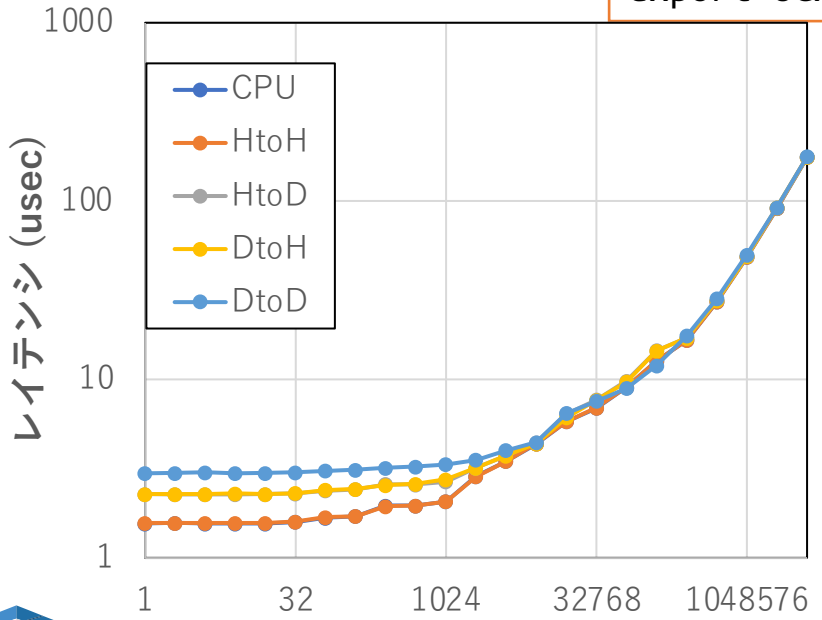
```
#!/bin/sh
case $OMPI_COMM_WORLD_LOCAL_RANK in
        0|1) export UCX_NET_DEVICES=mlx5_0:1 ;;
        2|3) export UCX_NET_DEVICES=mlx5_1:1 ;;
        4|5) export UCX_NET_DEVICES=mlx5_2:1 ;;
        6|7) export UCX_NET_DEVICES=mlx5_3:1 ;;
esac
export LOCAL_RANK=$OMPI_COMM_WORLD_LOCAL_RANK
export CUDA_VISIBLE_DEVICES=$LOCAL_RANK
numactl -l $*
```

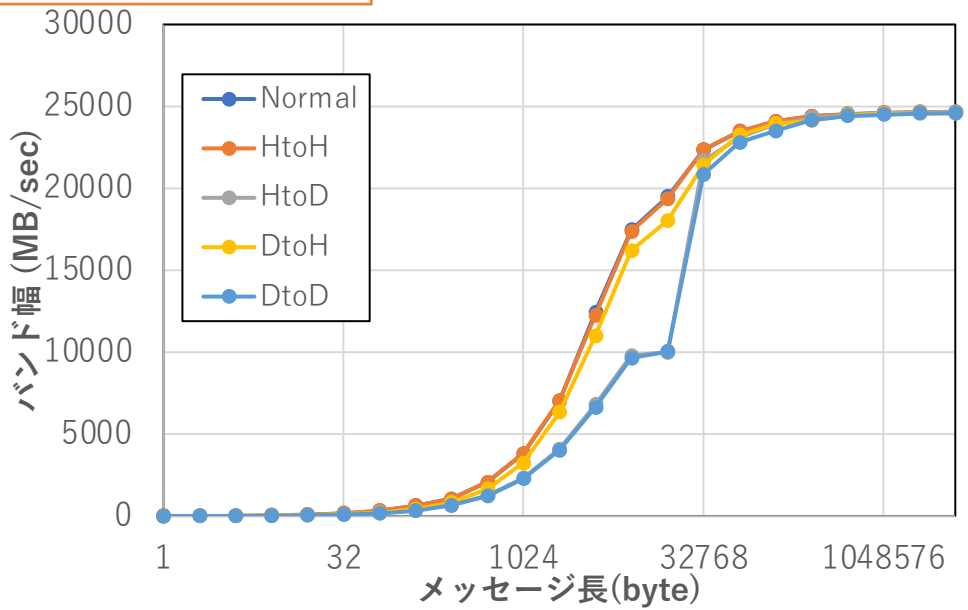
←必要に応じて

ノード間通信 (CUDA: 12.0, OpenMPI 4.1.5) NEW!!

- レイテンシ：osu_latency
  - HtoH: **1.57 us (CPUと同じ)**
  - DtoH, HtoDはほぼ同じ: **2.28 us**
  - DtoD: **2.98 us**
- バンド幅: osu_bw
  - 最大: **~25 GB/s**
  - **8KB~16KBの *toDが落ち込み**

```
export UCX_MAX_RNDV_RAILS=1
```

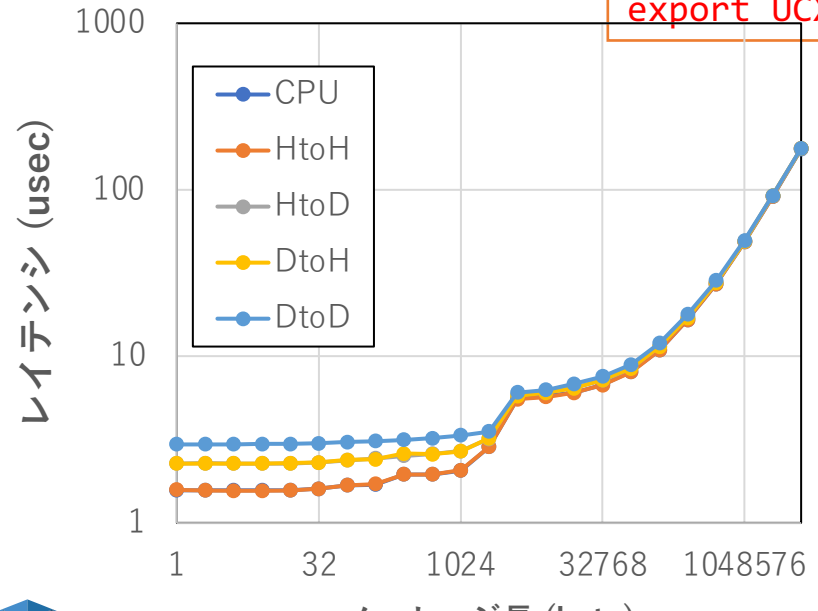




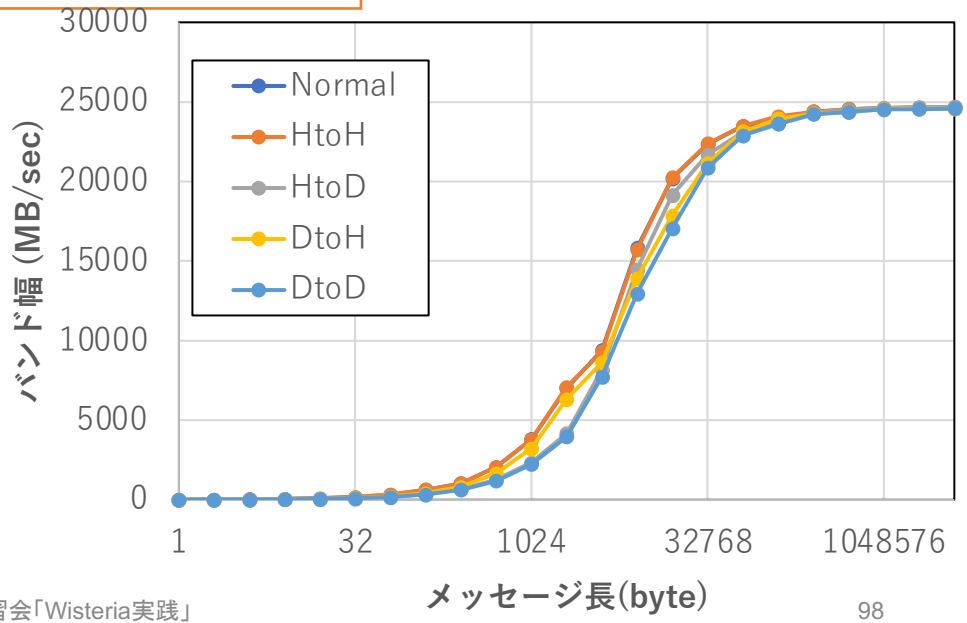
ノード間通信 (CUDA: 12.0, OpenMPI 4.1.5,パラメータ調整後)

- レイテンシ：osu_latency
  - HtoH: **1.57 us (CPUと同じ)**
  - DtoH, HtoDはほぼ同じ: **2.28 us**
  - DtoD: **2.98 us**

```
export UCX_MAX_RNDV_RAILS=1
export UCX_RNDV_THRESH=4K
```



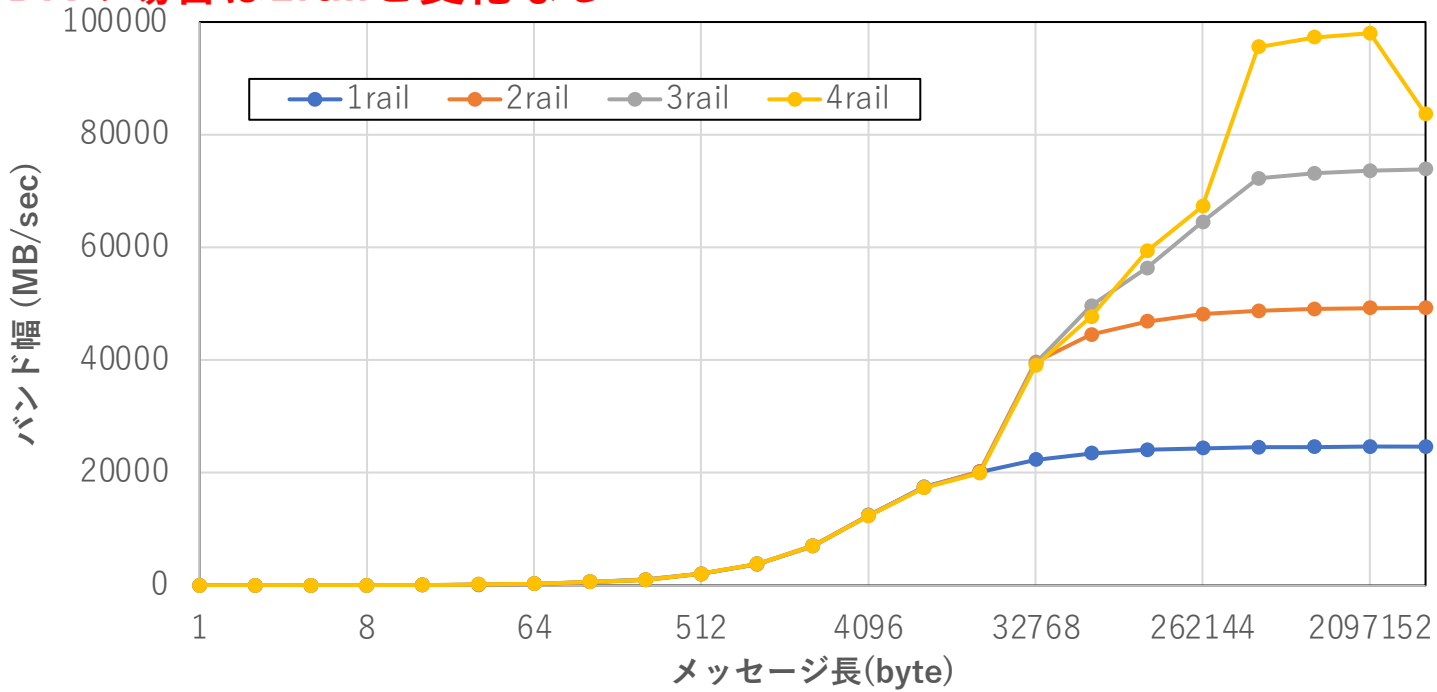
- バンド幅: osu_bw
  - 最大: **~25 GB/s**
  - ***toDの8KB~16KBの落ち込みは改善、*toHが悪化**



ノード間通信 (CUDA: 12.0, OpenMPI 4.1.5)

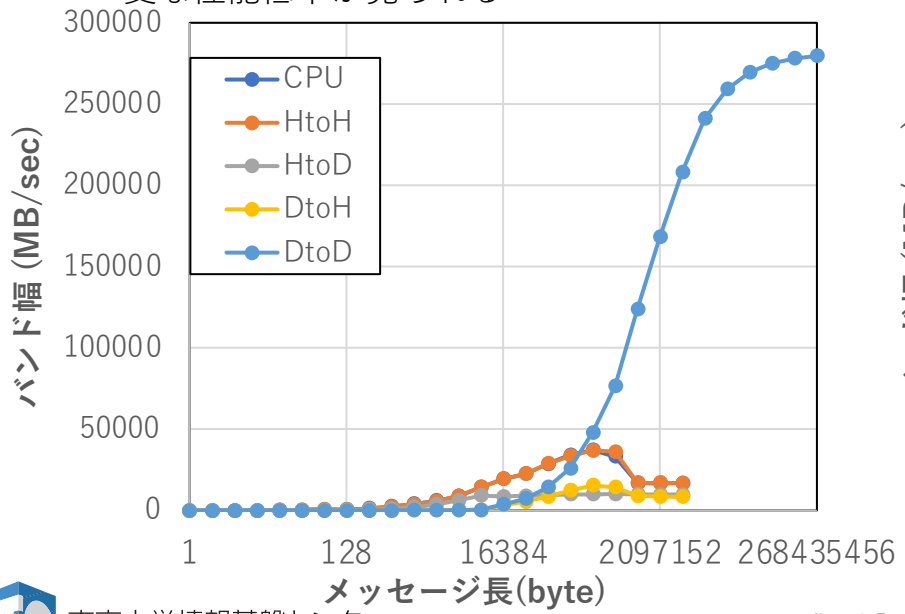
- バンド幅: osu_bw
  - 最大: **25, 49, 74, 98 GB/s(CUDA なし)**
    CUDAの場合は1railと変化なし

```
export UCX_MAX_RNDV_RAILS=1,2,3,4
export UCX_MAX_EAGER_RAILS=1,2,3,4
export UCX_RNDV_THRESH=16K
```



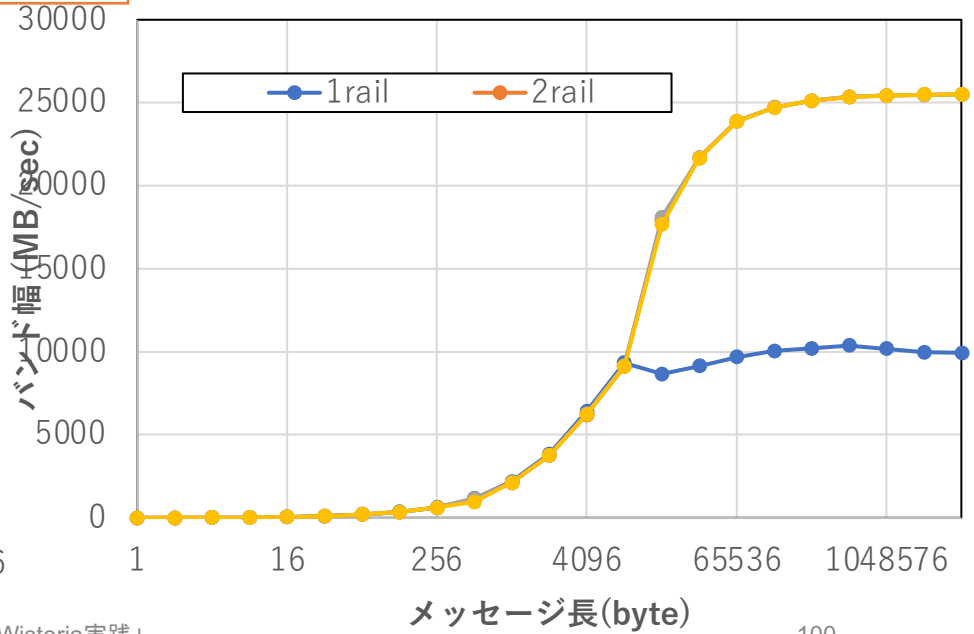
ノード内通信 (CUDA: 12.0, OpenMPI 4.1.5)

- バンド幅: osu_bw
  - デフォルトでInfiniBandを使う
    - メモリ経由より速い
  - DtoD: ~280 GB/sec, NVLink3 (300GB/s)の性能
  - DtoH, HtoDでは ~10, ~15GB/s 、HtoHでは ~36GB/s
  - 変な性能低下が見られる



- バンド幅: osu_bw
  - マルチレールが効く(2 railまで)
  - **HtoD**

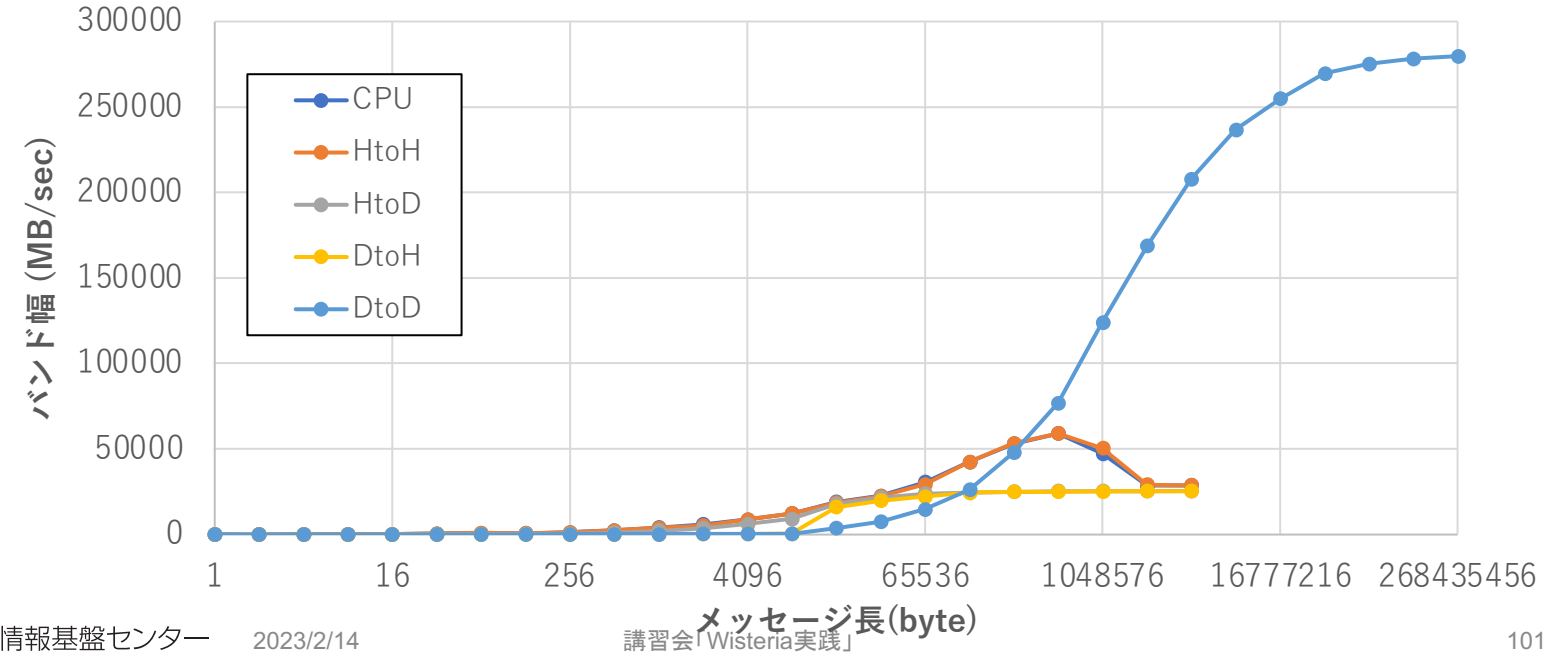
```
export UCX_MAX_RNDV_RAILS=1
```



ノード内通信 (CUDA: 12.0, OpenMPI 4.1.5)

- バンド幅: osu_bw
  - デフォルトでInfiniBandを使う
    - メモリ経由より速い
  - DtoD: ~280 GB/sec, NVLink3 (300GB/s)の性能
  - DtoH, HtoDでは ~10, ~15GB/s 、HtoHでは~36GB/s
  - 変な性能低下が見られる

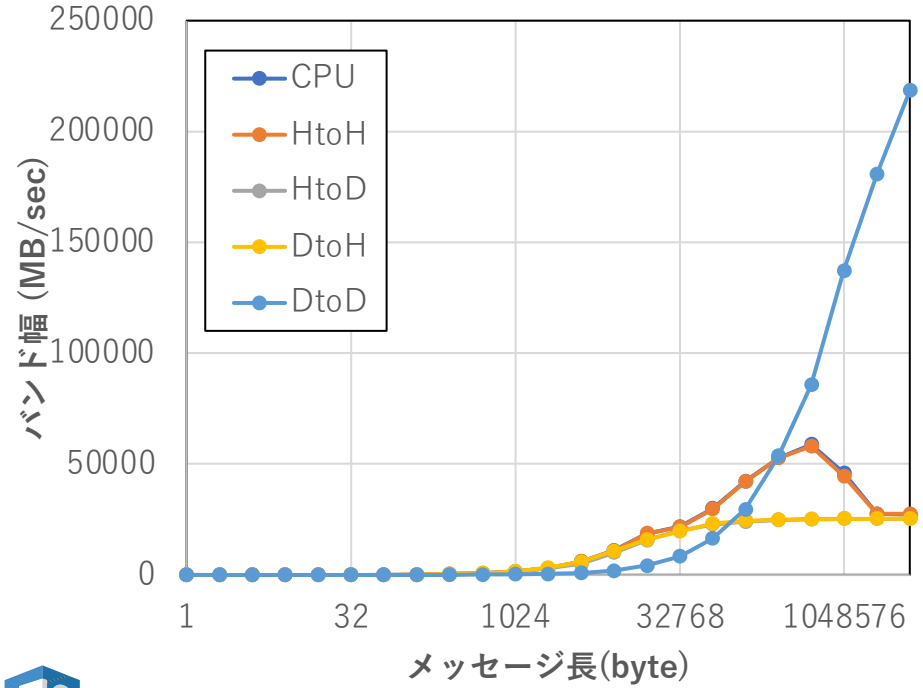
```
export UCX_MAX_RNDV_RAILS=2
```

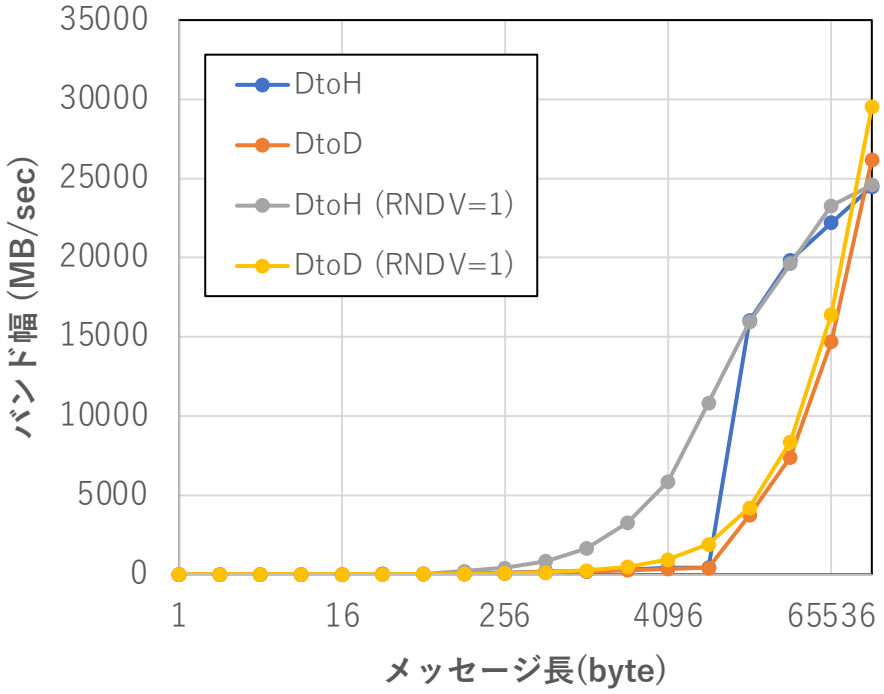


ノード内通信 (CUDA: 12.0, パラメータ調整)

- Dto*パターンのとき短メッセージで性能が出ない➔以下で改善
  - export UCX_RNDV_THRESH=1
  - 他のパターンでは副作用あり

export UCX_MAX_RNDV_RAILS=2





Wisteria/BDEC-01のストレージ

- FEFS (Fujitsu Exabyte File System)
- 並列ファイルシステム(4 rack)
  - **25 PB, 500 GB/s**
  - 4 MDS
  - OSS: DDN SFA7990XE x16
  - /work
- 高速ファイルシステム(2 rack)
  - **1 PB, 1 TB/s**
  - 2 MDS
  - OSS: DDN SFA400NVXE x16
    - NVMe SSD
  - /data/scratch: スクラッチ
  - /data/perm: 恒久

ストレージ性能

- メタデータ性能 (mdtest)
  - 共有FSに対し，複数のクライアントから，個別のディレクトリに対するファイル操作
    - 高速FSも並列FSと同等の性能
  - O: 4シェルフ 100*4=400 Gbps有効?
  - A: 8ノード 200*4*8=6.4 Tbps

	Odyssey 4x4x8ノード 1024プロセス	Aquarius 8ノード 128プロセス
File creation	63,596	53,856
File stat	120,498	281,756
File removal	26,720	92,472

- ファイル書き込み性能 (IOR)
  - 共有FSに対し独立なファイルの書き込み性能 (MiB)
    - O: 極力GIOノードを割り当て
    - A: 全ノード

	Odyssey		Aquarius	
高速FS	20x12x16 ノード確保 960 プロセス	455,372	45ノード 720 プロセス	322,115
並列FS	16x24x16 ノード確保 3072 プロセス	281,480	45ノード 720 プロセス	236,643

Singularityの利用

Singularity: コンテナ仮想化

Singularity Hub

shub://

docker Docker Hub

docker://

Singularity recipe

コンテナイメージ
ファイル

Production向け (Read only)

```
$ singularity build
container.img <source>
```

環境構築 (writable image)

```
$ sudo singularity build
--writable container.img
<source>
```

環境構築 (sandbox)

```
$ sudo singularity build
--sandbox container.img
<source>
```

コンテナイメージ
ファイル

container.img

コンテナ環境内で実行 (shell)

```
$ singularity shell
container.img
> python mnist.py
```

コンテナ環境のコマンドで実行(exec)

```
$ singularity exec
container.img python
mnist.py
```

定義された通りに実行(run)

```
$ singularity run
container.img
or
$ ./container.img
```

Singularity向けジョブスクリプト(1ノード)

```
#!/bin/bash
#PJM -L rg=tutorial-a
#PJM -L gpu=1
#PJM -g gt00
#PJM -L jobenv=singularity
#PJM -L elapse=00:15:00
#PJM -j

cd $PJM_O_WORKDIR
module load gcc cuda singularity
SIF=/work/gt00/share/tf-image.file
singularity exec $SIF which python
singularity exec $SIF python --version
singularity exec --nv --bind `pwd` --bind /work/share/MNIST $SIF python keras-tf-mnist.py ¥
>& keras-tf-mnist.log.$PJM_JOBID
```

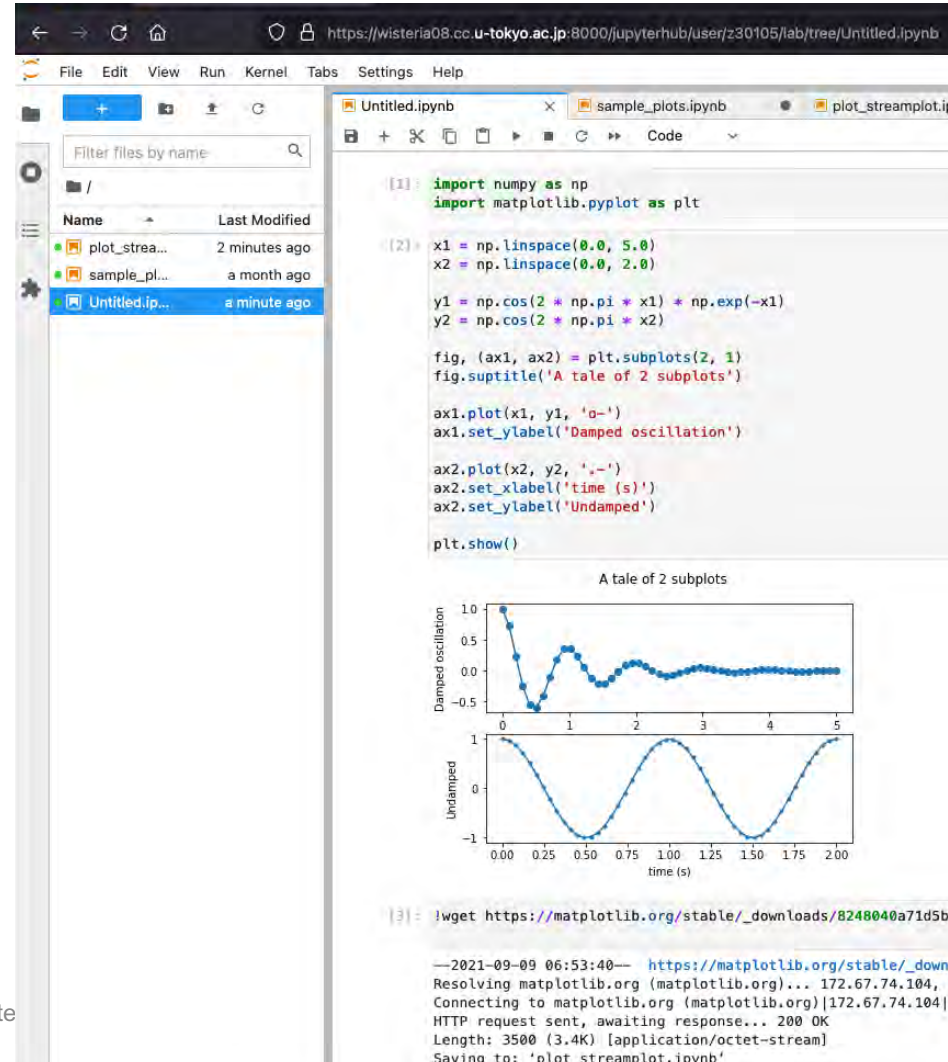
Singularity使用時に必要！！

コンテナの外にアクセスするパス毎に必要

Wisteria/BDEC-01の
便利な使い方：
JupyterHub, Guacamoleの紹介

JupyterHub

- JupyterNotebookなどの環境をポータルとして提供
  - 実はターミナル機能も使える
- https://wisteria08.cc.u-tokyo.ac.jp:8000/jupyterhub/
  - Windows: InternetExplorerでは不具合が起こることがある
  - MacOS: Safariではキー入力を受け付けないことがある、Firefox or Chrome推奨



JupyterNotebookからのジョブ投入

- 準備：

~/.notebook/template_job.sh を作成

```
#!/bin/sh
#PJM -L rscgrp=tutorial-a
#PJM -L gpu=1
```

- pysubコマンドでジョブ投入可能

```
[4] %cd /work/gt00/tABCDE/job_dir
[5] import os
[6] print(os.getcwd())
[7] f = open('./tmp.txt','a')
[8] （何か処理）
[9] f.close()
[10] !pysub -f Notebook.ipynb 5,6,7,8,9
```

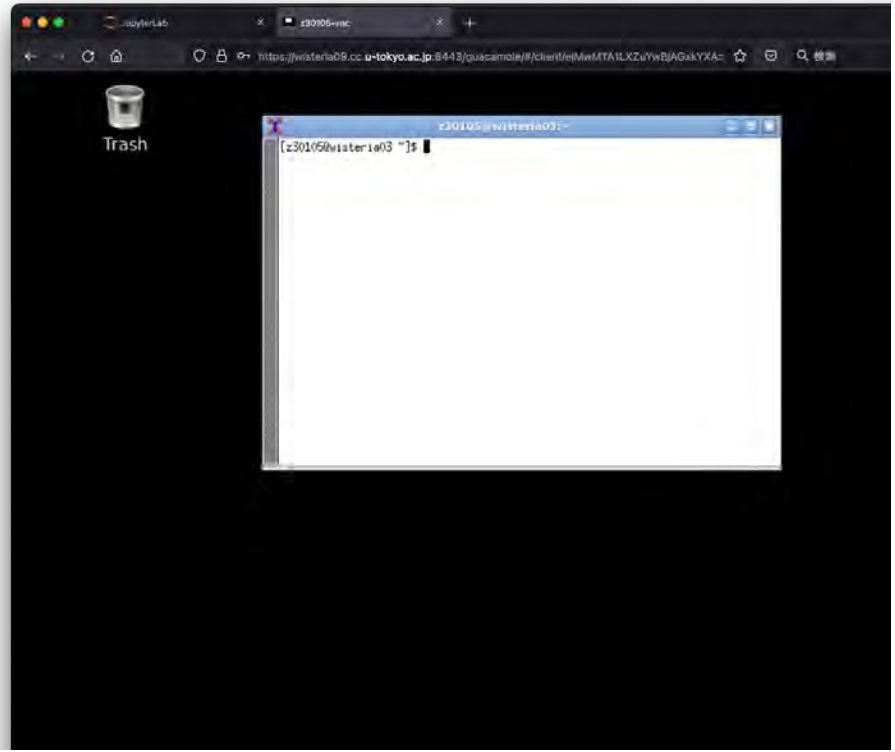
Pythonの利用

- 「Python環境の利用」スパコンニュース，2021年11月
  - https://www.cc.u-tokyo.ac.jp/public/VOL23/No6/12_202111Python.pdf
  - minicondaの利用方法
  - JupyterHubのカスタマイズ
  - Singularityの利用

Apache Guacamole

ブラウザ上でリモートデスクトップ実現

- VNC (Virtual Network Computing)のクライアント
  - X-Windowクライアントを手元に飛ばすよりずっと高速
- https://wisteria09.cc.u-tokyo.ac.jp:8443/guacamole/



Apache Guacamoleの準備

- ログインノードでvncserver起動(vnc用独自パスワードの設定が求められるかも)

```
$ vncserver
```

- 起動したポート番号確認: 5900+番号=> 5903

```
$ vncserver -list
X DISPLAY #     PROCESS ID
:3              891292
```

- 接続情報登録: set_vncserver -h 起動ホスト名 -p 起動ポート番号

```
$ set_vncserver -h wisteria03 -p 5903
Enter LDAP Password: ポータルのパスワード
modifying entry "cn=z30105-vnc,…”
```

- DISPLAY環境変数の設定

```
$ export DISPLAY=:3
```

お疲れ様でした！
アンケートにご協力お願いします

- https://forms.office.com/r/LW6RCqt7Bp