第106回　お試しアカウント付き 並列プログラミング講習会 「MPI上級」

東京大学情報基盤センター

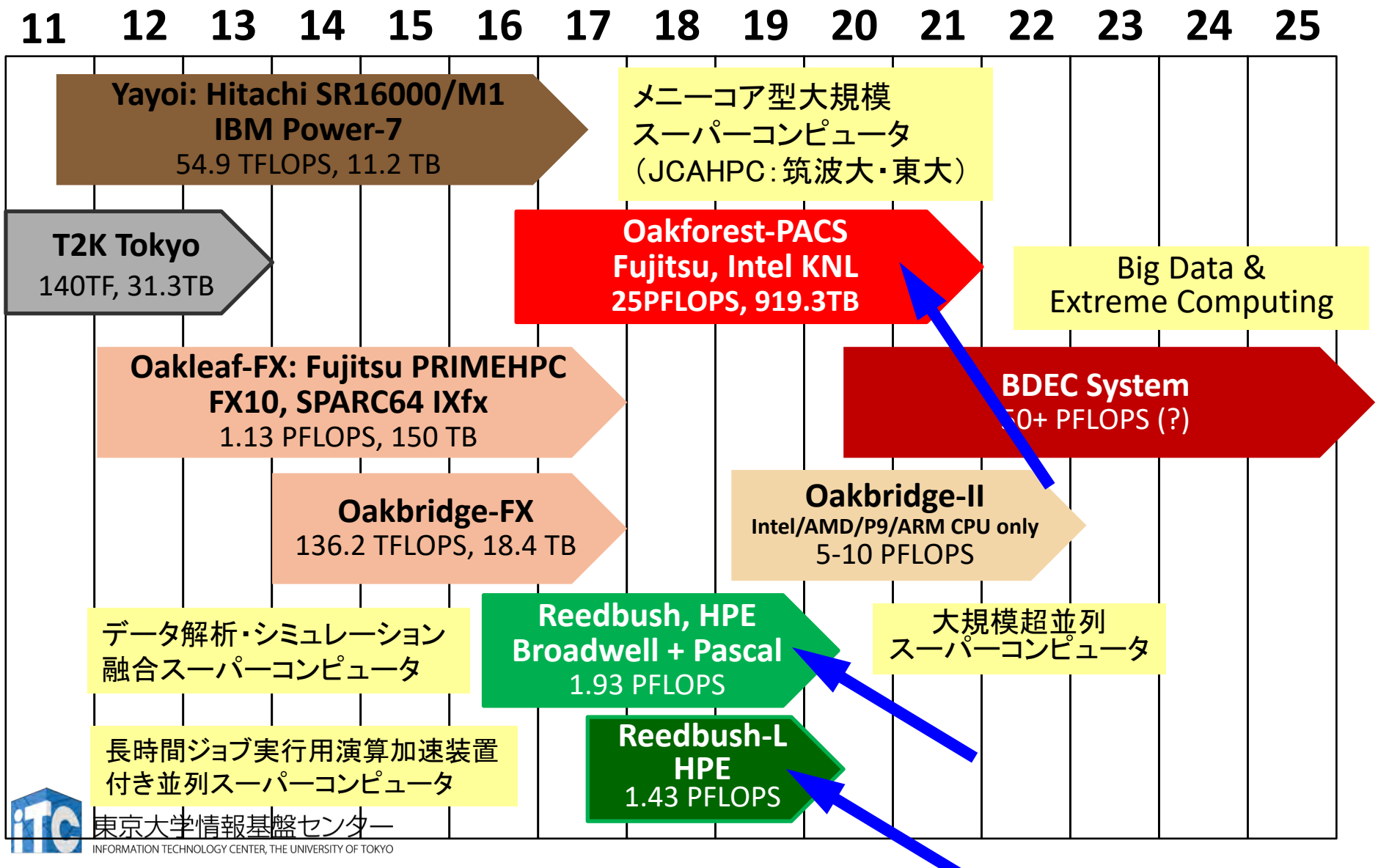
内容に関するご質問は
hanawa @ cc.u-tokyo.ac.jp
まで、お願いします。

講習会概略

- 開催日：
  2018年10月29日（月） 10：00 - 17：45
- 場所：東京大学情報基盤センター　4階　413遠隔会義室
- 講師：塙　敏博（スーパーコンピューティング研究部門・准教授）
- 講習会プログラム：
- 10月29日（月）
  - 9：30 - 10：00　受付
  - 10：00 - 12：00　Oakforest-PACSログイン、MPI概要、Oakforest-PACSで使えるMPI実装の紹介、ノンブロッキング通信、演習
  - 13：30 - 14：45　派生データ型、MPI-IO、演習
  - 15：00 - 16：15　コミュニケータ、マルチスレッドとMultiple-Endpoint、演習
  - 16：30 - 17：45　片側通信、演習

東大センターのスパコン

2基の大型システム，6年サイクル（だった）



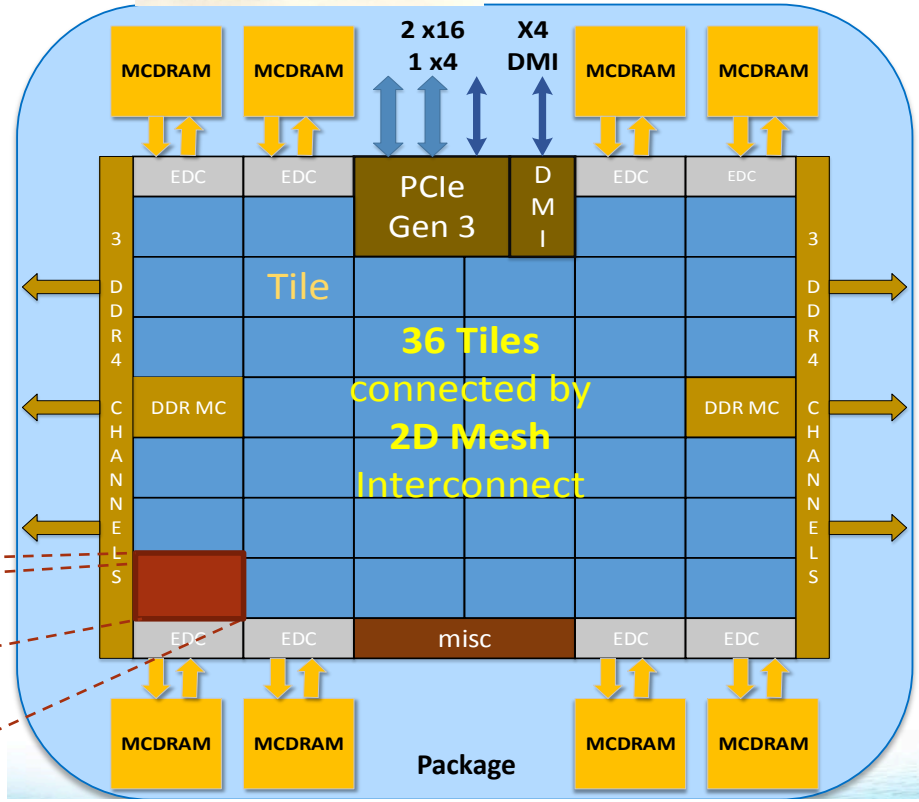
Oakforest-PACS (OFP)

- 2016年12月1日稼働開始
- 8,208 Intel Xeon/Phi (KNL)、ピーク性能25PFLOPS
  - 富士通が構築
- **TOP 500　6位（国内1位），HPCG 3位（国内2位）（2016年11月）**
- **最先端共同HPC 基盤施設(JCAHPC: Joint Center for Advanced High Performance Computing)**
  - 筑波大学計算科学研究センター
  - 東京大学情報基盤センター
    - 東京大学柏キャンパスの東京大学情報基盤センター内に、両機関の教職員が中心となって設計するスーパーコンピュータシステムを設置し，最先端の大規模高性能計算基盤を構築・運営するための組織
  - http://jcahpc.jp

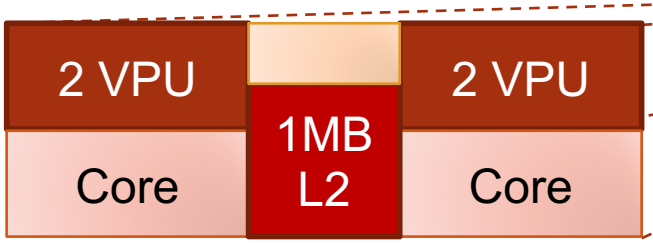
Oakforest-PACS 計算ノード

- Intel Xeon Phi (Knights Landing)
  - 1ノード1ソケット, 68コア
- MCDRAM: オンパッケージの高バンド幅メモリ16GB + DDR4メモリ 16GBx6 = 16 + 96 GB

HotChips27
KNLスライドより



MCDRAM: 490GB/秒以上 （実測）DDR4: 115.2 GB/秒
=(8Byte×2400MHz×6 channel)



Oakforest-PACS利用の準備

配布資料「Oakforest-PACS利用の手引き」参照

ユーザ名の確認

- 本講習会でのユーザー名

利用者番号： t00631～

利用グループ： gt00

11/29 9:00まで有効

東大スーパーコンピュータの概略

2（または3,4）システム運用中

- Oakleaf-FX (富士通 PRIMEHPC FX10)
  - 1.135 PF, 京コンピュータ商用版, 2012年4月 ～ 2018年3月
- Oakbridge-FX (富士通 PRIMEHPC FX10)
  - 136.2 TF, 長時間実行用（168時間）, 2014年4月 ～ 2018年3月

- Reedbush (HPE, Intel BDW + NVIDIA P100 (Pascal))
  - データ解析・シミュレーション融合スーパーコンピュータ
    - 2016-Jun.2016年7月～2020年6月
  - 東大情基セ初のGPU搭載システム
  - Reedbush-U: CPU only, 420 nodes, 508 TF (2016年7月)
  - Reedbush-H: 120 nodes, 2 GPUs/node: 1.42 PF (2017年3月)
  - Reedbush-L: 64 nodes, 4 GPUs/node: 1.43 PF (2017年10月)

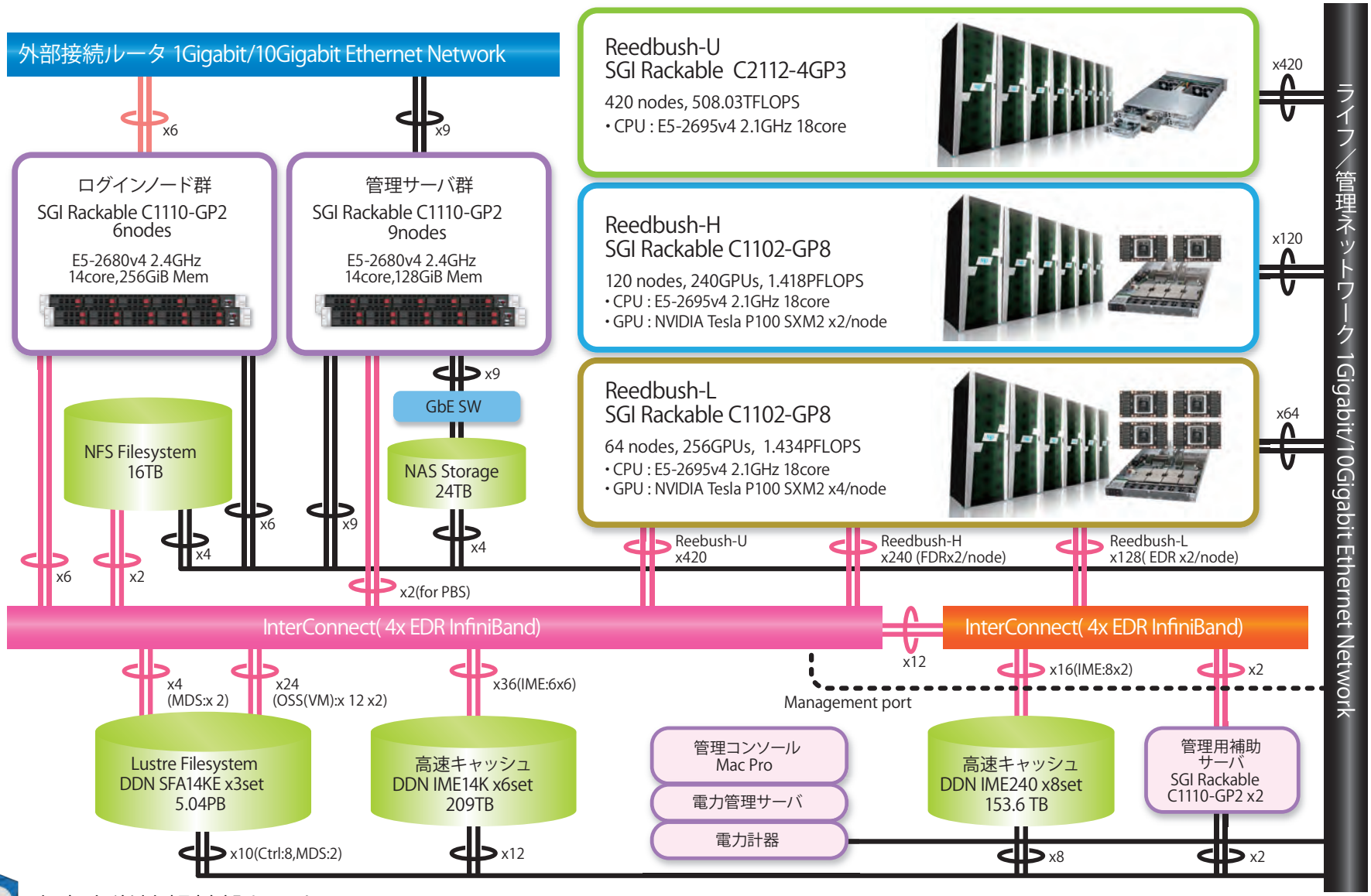
- **Oakforest-PACS (OFP)** (富士通, Intel Xeon Phi (KNL))
  - JCAHPC (筑波大CCS＆東大ITC)
  - 25 PF, 世界第12位 (2018年6月) (日本第2位)
  - Omni-Path アーキテクチャ, DDN IME (Burst Buffer)

Reedbushシステム

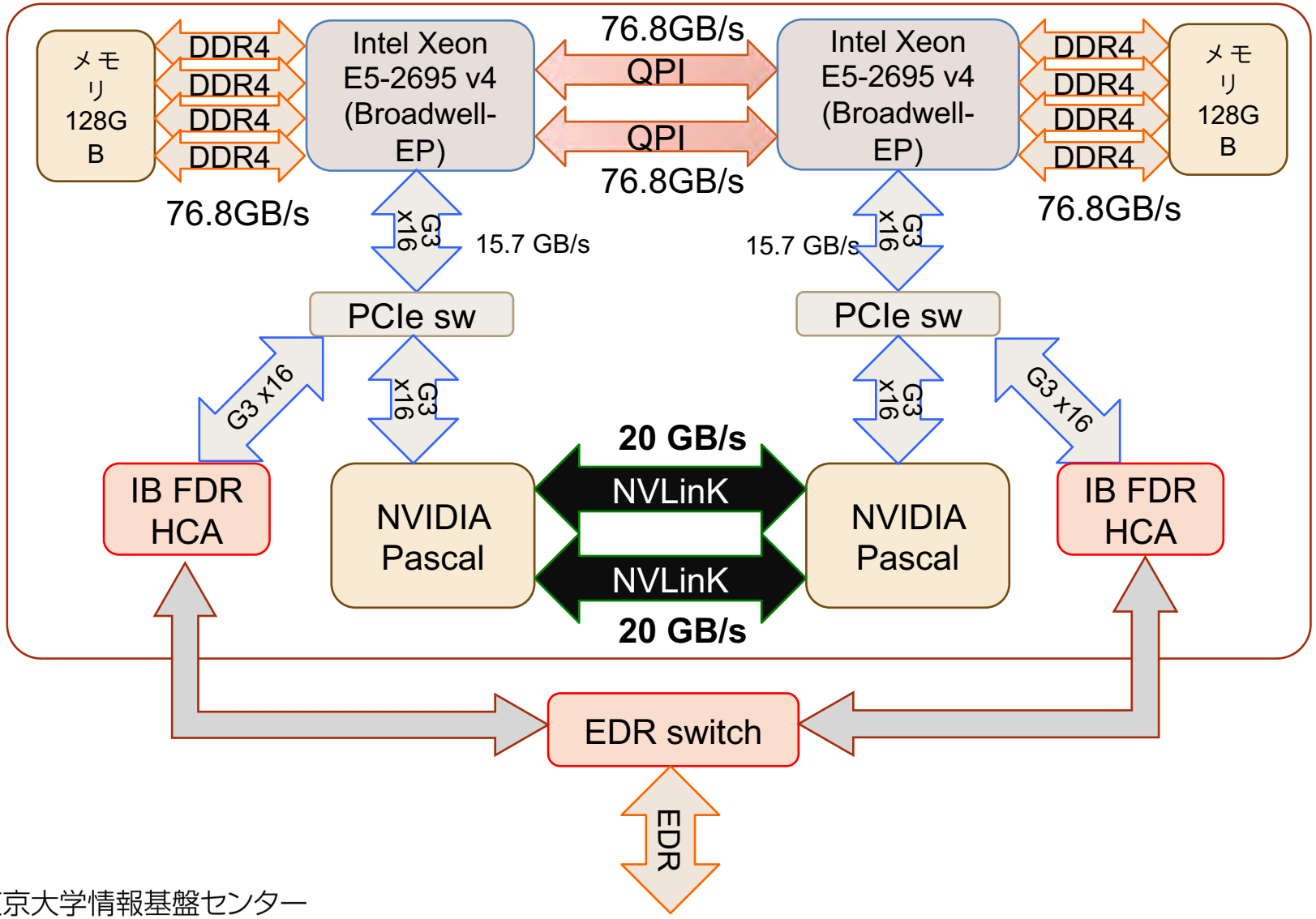
Top500: RB-L 291位@Nov. 2017
RB-H 203位@Jun. 2017
RB-U 361位@Nov. 2016
Green500: RB-L 11位@Nov. 2017
RB-H 11位@Jun. 2017

Reedbush-U
2016年7月1日 試験運転開始
2016年9月1日 正式運用開始
Reedbush-H
2017年3月1日 試験運転開始
2017年4月3日 正式運用開始
Reedbush-L
2017年10月2日 試験運転開始
2017年11月1日 正式運用開始

東京大学情報基盤センター　Reedbushシステム



Reedbush-Hノードのブロック図



Reedbushのサブシステム

	Reedbush-U	Reedbush-H	Reedbush-L
CPU/node	Intel Xeon E5-2695v4 (Broadwell-EP, 2.1GHz, 18core) x 2 sockets (1.210 TF), 256 GiB (153.6GB/sec)		
GPU	-	NVIDIA Tesla P100 (Pascal, 5.3TF, 720GB/sec, 16GiB)	
Infiniband	EDR	FDR×2ch	EDR×2ch
ノード数	420	120	64
GPU数	-	240 (=120×2)	256 (=64×4)
ピーク性能 (TFLOPS)	509	1,417 (145 + 1,272)	1,433 (76.8 + 1,358)
メモリバンド幅 (TB/sec)	64.5	191.2 (18.4+172.8)	194.2 (9.83+184.3)
運用開始	2016.07	2017.03	2017.10

Oakforest-PACS （OFP）

- 2016年12月1日稼働開始
- 8,208 Intel Xeon/Phi (KNL), ピーク性能25PFLOPS
  - 富士通が構築
- **TOP 500 12位（国内2位）， HPCG 7位（国内2位） （2018年6月）**
- **最先端共同HPC 基盤施設(JCAHPC: Joint Center for Advanced High Performance Computing)**
  - 筑波大学計算科学研究センター
  - 東京大学情報基盤センター
  - 東京大学柏キャンパスの東京大学情報基盤センター内に、両機関の教職員が中心となって設計するスーパーコンピュータシステムを設置し，最先端の大規模高性能計算基盤を構築・運営するための組織
  - http://jcahpc.jp

JCAHPC

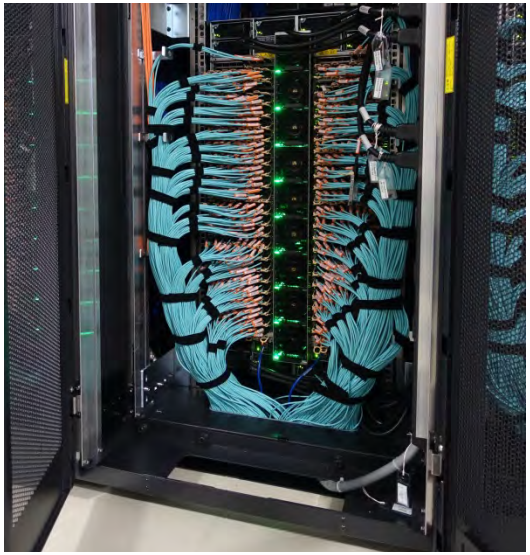
東京大学情報基盤センター INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

東京大学 THE UNIVERSITY OF TOKYO

筑波大学 University of Tsukuba

Oakforest-PACSの特徴 (1/2)

- 計算ノード
  - 1ノード 68コア，3TFLOPS×8,208ノード＝ 25 PFLOPS
  - メモリ（MCDRAM（高速，16GB）＋DDR4（低速，96GB））
- ノード間通信
  - フルバイセクションバンド幅を持つFat-Treeネットワーク
  - 全系運用時のアプリケーション性能に効果，多ジョブ運用
  - Intel Omni-Path Architecture

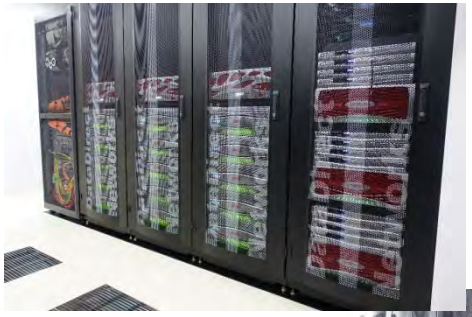


Oakforest-PACS の仕様

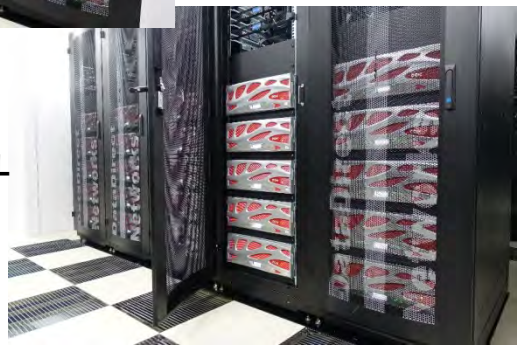
総ピーク演算性能			25 PFLOPS
ノード数			8,208
計算ノード	Product		富士通 PRIMERGY CX600 M1 (2U) + CX1640 M1 x 8node
	プロセッサ		Intel® Xeon Phi™ 7250（開発コード: Knights Landing）68 コア、1.4 GHz
	メモリ	高バンド幅	16 GB, MCDRAM, 実効 490 GB/sec
		低バンド幅	96 GB, DDR4-2400, ピーク 115.2 GB/sec
相互結合網	Product		Intel® Omni-Path Architecture
	リンク速度		100 Gbps
	トポロジ		フルバイセクションバンド幅Fat-tree網

Oakforest-PACS の特徴（2 / 2）

- ファイルI/O
  - 並列ファイルシステム:
    Lustre 26PB
  - ファイルキャッシュシステム（DDN IME）：
    1TB/secを超える実効性能,
    約1PB
    - 計算科学・ビッグデータ解析・機械学習にも貢献
- 消費電力
  - Green 500でも世界6位 (2016.11)
  - Linpack： 2.72 MW
    - 4,986 MFLOPS/W（OFP）
    - 830 MFLOPS/W（京）



並列ファイルシステム



ファイルキャッシュシステム

ラック当たり120ノードの高密度実装

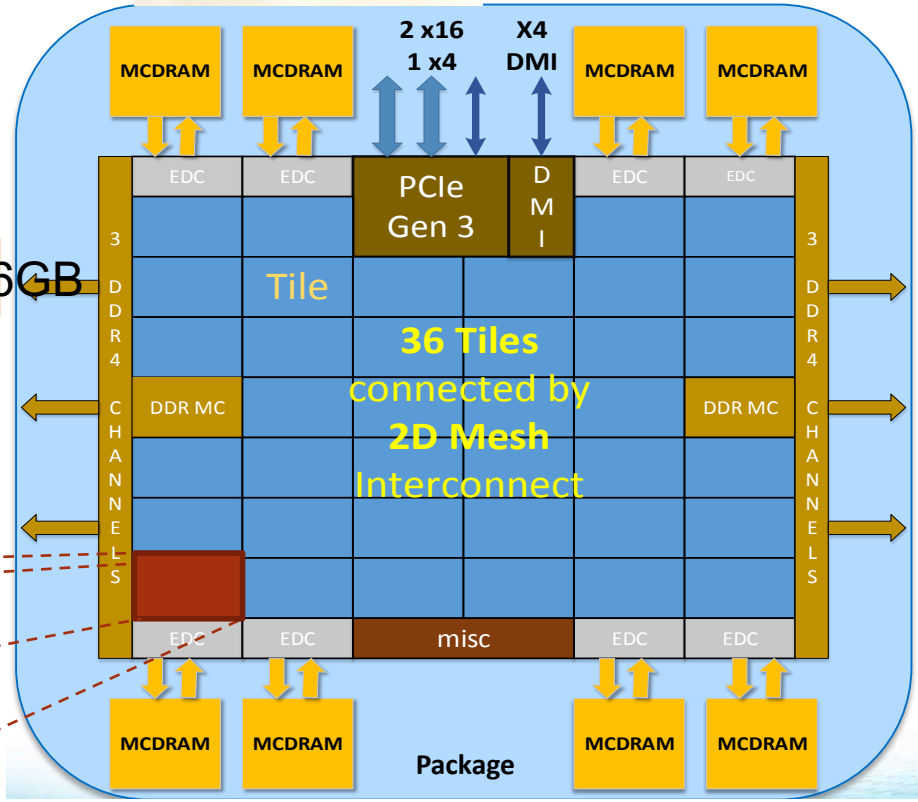
Oakforest-PACS の仕様（続き）

並列ファイルシステム	Type	Lustre File System
	総容量	26.2 PB
	Product	DataDirect Networks SFA14KE
	総バンド幅	500 GB/sec
高速ファイルキャッシュシステム	Type	Burst Buffer, Infinite Memory Engine (by DDN)
	総容量	940 TB (NVMe SSD, パリティを含む)
	Product	DataDirect Networks IME14K
	総バンド幅	1,560 GB/sec
総消費電力		4.2MW（冷却を含む）
総ラック数		102

Oakforest-PACS 計算ノード

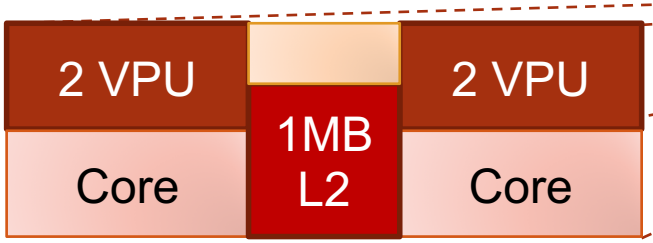
- Intel Xeon Phi (Knights Landing)
  - 1ノード1ソケット
- MCDRAM: オンパッケージの高バンド幅メモリ16GB + DDR4メモリ

HotChips27
KNLスライドより

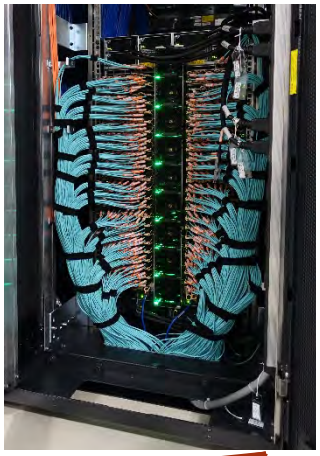


ソケット当たりメモリ量：16GB×6＝96GB

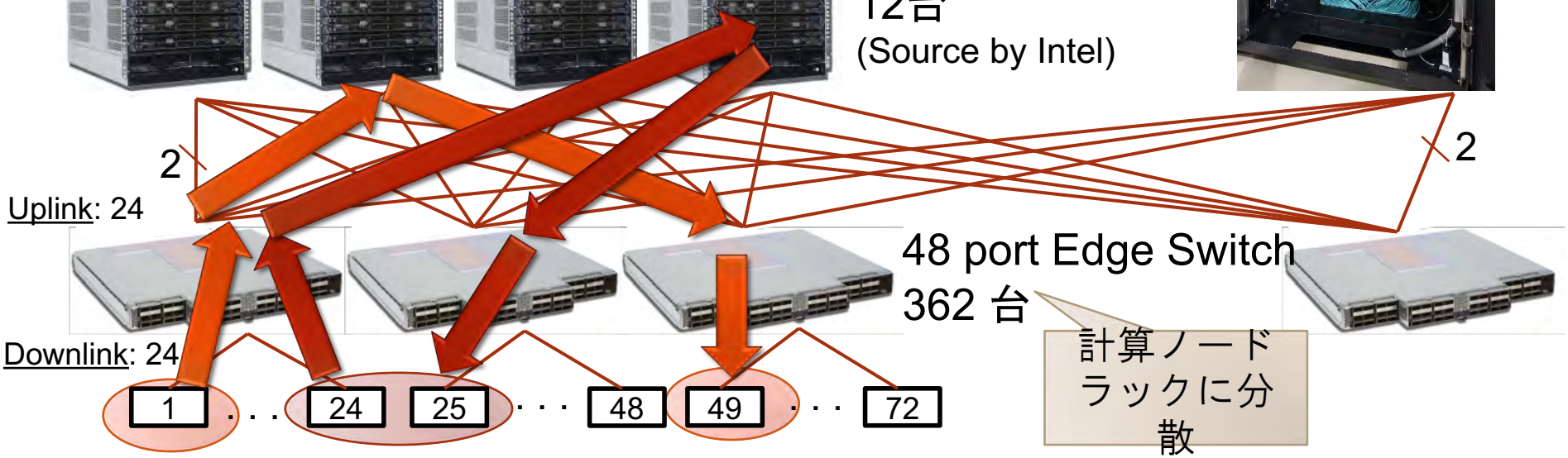
MCDRAM: 490GB/秒以上 （実測）DDR4: 115.2 GB/秒
=(8Byte×2400MHz×6 channel)



Oakforest-PACS: Intel Omni-Path Architecture によるフルバイセクションバンド幅Fat-tree網



768 port Director Switch
12台
(Source by Intel)



コストはかかるがフルバイセクションバンド幅を維持
- システム全系使用時にも高い並列性能を実現
- 柔軟な運用：ジョブに対する計算ノード割り当ての自由度が高い

51th TOP500 List (June, 2018)

R_{max}: Performance of Linpack (TFLOPS)
R_{peak}: Peak Performance (TFLOPS),
Power: kW

http://www.top500.org/

	Site	Computer/Year Vendor	Cores	R_{max} (TFLOPS)	R_{peak} (TFLOPS)	Power (kW)
1	**Summit, 2018, USA** DOE/SC/Oak Ridge National Laboratory	IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband	2,282,544	122,300 (= 122.3 PF)	187,659	8,806
2	**Sunway TaihuLight, 2016, China** National Supercomputing Center in Wuxi	Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway	10,649,600	93,015	125,436	15,371
3	**Sieera, 2018, USA** DOE/NNSA/LLNL	IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband	1,572,480	71,610	119,194	
4	**Tianhe-2A, 2018, China** National Super Computer Center in Guangzhou	TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000	4,981,760	61,445	100,679	18,482
5	**ABCI (AI Bridging Cloud Infrastructure), 2018, Japan** National Institute of Advanced Industrial Science and Technology (AIST)	PRIMERGY CX2550 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR	391,680	19,880	32,577	1,649
6	**Piz Daint, 2017, Switzerland** Swiss National Supercomputing Centre (CSCS)	Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100	361,760	19,590	25,326	2,272
7	**Titan, 2012, USA** DOE/SC/Oak Ridge National Laboratory	Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x	560,640	17,590	27,113	8,209
8	**Sequoia, 2011, USA** DOE/NNSA/LLNL	BlueGene/Q, Power BQC 16C 1.60 GHz, Custom	1,572,864	17,173	20,133	7,890
9	**Trinity, 2017, USA** DOE/NNSA/LANL/SNL	Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect	979,968	14,137	43,903	3,844
10	**Cori, 2016, Japan** DOE/SC/LBNL/NERSC	Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect	622,336	14,016	27,881	3,939
12	**Oakforest-PACS, 2016, Japan** Joint Center for Advanced High Performance Computing	PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path	556,104	13,556	24,913	2,719

HPCG Ranking (June, 2018)

	Computer	Cores	HPL Rmax (Pflop/s)	TOP500 Rank	HPCG (Pflop/s)	Peak
1	**Summit**	2,392,000	122.300	1	2.926	1.5%
2	**Sierra**	835,584	71.610	3	1.796	1.5%
3	**K computer**	705,024	10.510	16	0.603	5.3%
4	**Trinity**	979,072	14.137	9	0.546	1.8%
5	**Piz Daint**	361,760	19.590	6	0.486	1.9%
6	**Sunway TaihuLight**	10,649,600	93.015	2	0.481	0.4%
7	**Oakforest-PACS**	557,056	13.555	12	0.385	1.5%
8	**Cori**	632,400	13.832	10	0.355	1.3%
9	**Tera-1000-2**	522,240	11.965	14	0.334	1.4%
10	**Sequoia**	1,572,864	17.173	8	0.330	1.6%

http://www.hpcg-benchmark.org/

Green 500 Ranking (SC16, November, 2016)

	Site	Computer	CPU	HPL Rmax (Pflop/s)	TOP500 Rank	Power (MW)	GFLOPS/W
1	NVIDIA Corporation	DGX SATURNV	NVIDIA DGX-1, Xeon E5-2698v4 20C 2.2GHz, Infiniband EDR, NVIDIA Tesla P100	3.307	28	0.350	9.462
2	Swiss National Supercomputing Centre (CSCS)	Piz Daint	Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100	9.779	8	1.312	7.454
3	RIKEN ACCS	Shoubu	ZettaScaler-1.6 etc.	1.001	116	0.150	6.674
4	National SC Center in Wuxi	Sunway TaihuLight	Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway	93.01	1	15.37	6.051
5	SFB/TR55 at Fujitsu Tech. Solutions GmbH	QPACE3	PRIMERGY CX1640 M1, Intel Xeon Phi 7210 64C 1.3GHz, Intel Omni-Path	0.447	375	0.077	5.806
6	JCAHPC	Oakforest-PACS	PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path	1.355	6	2.719	4.986
7	DOE/SC/Argonne National Lab.	Theta	Cray XC40, Intel Xeon Phi 7230 64C 1.3GHz, Aries interconnect	5.096	18	1.087	4.688
8	Stanford Research Computing Center	XStream	Cray CS-Storm, Intel Xeon E5-2680v2 10C 2.8GHz, Infiniband FDR, Nvidia K80	0.781	162	0.190	4.112
9	ACCMS, Kyoto University	Camphor 2	Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect	3.057	33	0.748	4.087
10	Jefferson Natl. Accel. Facility	SciPhi XVI	KOI Cluster, Intel Xeon Phi 7230 64C 1.3GHz, Intel Omni-Path	0.426	397	0.111	3.837

http://www.top500.org/

http://www.top500.org/

Green 500 Ranking (June, 2018)

	TOP 500 Rank	System	Cores	HPL Rmax (Pflop/s)	Power (MW)	GFLOPS/W
1	359	Shoubu system B, Japan	794,400	858.	47	18.404
2	419	Suiren2, Japan	762,624	798.	47	16.835
3	385	Sakura, Japan	794,400	825.	50	16.657
4	227	DGX SaturnV Volta, USA	22,440	1,070.	97	15.113
5	1	Summit, USA	2,282,544	122,300.	8,806	13.889
6	19	TSUBAME3.0, Japan	135,828	8,125.	792	13.704
7	287	AIST AI Cloud, Japan	23,400	961.	76	12.681
8	5	ABCI, Japan	391,680	19,880.	1,649	12.054
9	255	MareNostrum P9 CTE, Spain	19,440	1,018.	86	11.865
10	171	RAIDEN GPU, Japan	35,360	1,213.	107	11.363
13	411	Reedbush-L, U.Tokyo, Japan	16,640	806.	79	10.167
19	414	Reedbush-H, U.Tokyo, Japan	17,760	802.	94	8.575

IO 500 Ranking (June, 2018)

	Site	Computer	File system	Client nodes	IO500 Score	BW (GiB/s)	MD (kIOP/s)
1	JCAHPC, Japan	Oakforest-PACS	DDN IME	2048	137.78	560.10	33.89
2	KAUST, Saudi	Shaheen2	Cray DataWarp	1024	77.37	496.81	12.05
3	KAUST, Saudi	Shaheen2	Lustre	1000	41.00	54.17	31.03
4	JSC, Germany	JURON	BeeGFS	8	35.77	14.24	89.81
5	DKRZ, Germany	Mistral	Lustre2	100	32.15	22.77	45.39
6	IBM, USA	Sonasad	Spectrum Scale	10	24.24	4.57	128.61
7	Fraunhofer, Germany	Seislab	BeeGFS	24	16.96	5.13	56.14
8	DKRZ, Germany	Mistral	Lustre1	100	15.47	12.68	18.88
9	Joint Institute for Nuclear Research, Russia	Govorun	Lustre	24	12.08	3.34	43.65
10	PNNL, USA	EMSL Cascade	Lustre	126	11.12	4.88	25.33

http://www.io500.org/

東大情報基盤センターOakforest-PACSスーパーコンピュータシステムの料金表（2018年4月1日 ）

- パーソナルコース
  - 100,000円　：　1口8ノード(基準) 3口まで、最大2048ノードまで
    トークン：2ノード x 24時間 x 360日分（1口）
- グループコース
  - 400,000円 (企業 480,000円)　：　1口 8ノード（基準）、最大2048ノードまで
    トークン：8ノード x 24時間 x 360日分 （1口）
- 以上は、「トークン制」で運営
  - 基準ノード数までは、トークン消費係数が1.0
  - 基準ノード数を超えると、超えた分は、消費係数が2.0になる
  - 大学等のユーザはReedbushとの相互トークン移行も可能

東大情報基盤センターReedbushスーパーコンピュータシステムの料金表（2018年4月1日 ）

- パーソナルコース
  - 150,000円 ：RB-U: 4ノード（基準）、最大128ノードまで
    RB-H: 1ノード（基準、係数はUの2.5倍）、最大 32ノードまで
    RB-L: 1ノード（基準、係数はUの4倍）、最大 16ノードまで
- グループコース
  - 300,000円：RB-U 1口 4ノード（基準）、最大128ノードまで、
    RB-H: 1ノード（基準、係数はUの2.5倍）、最大 32ノードまで
    RB-L: 1ノード（基準、係数はUの4倍）、最大 16ノードまで
  - 企業 RB-Uのみ 360,000円 ： 1口 4ノード（基準）、最大128ノードまで
  - 企業 RB-Hのみ 216,000円 ： 1口 1ノード（基準）、最大32ノードまで
  - 企業 RB-Lのみ 360,000円 ： 1口 1ノード（基準）、最大16ノードまで
- 以上は、「トークン制」で運営
  - 申し込みノード数×360日×24時間の「トークン」が与えられる
  - 基準ノードまでは、トークン消費係数が1.0 (Hはその2.5倍, Lは4倍)
  - 基準ノードを超えると、超えた分は、消費係数がさらに2倍になる
  - 大学等のユーザはOakforest-PACSとの相互トークン移行も可能
  - ノード固定もあり (Reedbush-U, L)

トライアルユース制度について

- 安価に当センターのReedbush-U/H/L, Oakforest-PACSシステムが使える「無償トライアルユース」および「有償トライアルユース」制度があります。
  - アカデミック利用
    - パーソナルコース（1～3ヶ月）（RB-U: 最大16ノード, RB-H: 最大4ノード、RB-L: 最大4ノード、OFP: 最大16ノード）
    - グループコース（1～9ヶ月）（RB-U: 最大128ノード、RB-H: 最大32ノード、RB-L: 最大16ノード、OFP: 最大2048ノード）
  - 企業利用
    - パーソナルコース（１～３ヶ月）（RB-U: 最大16ノード, RB-H: 最大4ノード、RB-L: 最大4ノード、OFP: 最大16ノード）
      本講習会の受講が必須、審査無
    - グループコース
      - 無償トライアルユース：（１ヶ月～３ヶ月）：無料（RB-U: 最大128ノード、RB-H: 最大32ノード、OFP: 最大2048ノード）
      - 有償トライアルユース：（１ヶ月～最大通算９ヶ月）、有償（計算資源は無償と同等）
      - スーパーコンピュータ利用資格者審査委員会の審査が必要（年2回実施）
    - 双方のコースともに、簡易な利用報告書の提出が必要

スーパーコンピュータシステムの詳細

- 以下のページをご参照ください
  - 利用申請方法
  - 運営体系
  - 料金体系
  - 利用の手引

などがご覧になれます。

https://www.cc.u-tokyo.ac.jp/guide/

MPI (Message Passing Interface)

おさらいも兼ねて

MPIの特徴

- メッセージパッシング用のライブラリ規格の１つ
  - メッセージパッシングのモデルである
  - コンパイラの規格、特定のソフトウエアやライブラリを指すものではない！
- 分散メモリ型並列計算機で並列実行に向く
- 大規模計算が可能
  - １プロセッサにおけるメモリサイズやファイルサイズの制約を打破可能
  - プロセッサ台数の多い並列システム（Massively Parallel Processing (MPP)システム）を用いる実行に向く
    - １プロセッサ換算で膨大な実行時間の計算を、短時間で処理可能
  - 移植が容易
    - API（Application Programming Interface）の標準化
- スケーラビリティ、性能が高い
  - 通信処理をユーザが記述することによるアルゴリズムの最適化が可能
  - プログラミングが難しい（敷居が高い）

MPIの経緯（これまで）

- MPIフォーラム（http://www.mpi-forum.org/）が仕様策定
  - 1994年5月 1.0版（MPI-1）
  - 1995年6月 1.1版
  - 1997年7月 1.2版、 および 2.0版（MPI-2）
  - 2008年5月 1.3版、 2008年6月 2.1版
  - 2009年9月 2.2版
    - 日本語版 http://www.pccluster.org/ja/mpi.html
- MPI-2 では、以下を強化：
  - 並列I/O
  - C++、Fortran 90用インターフェース
  - 動的プロセス生成/消滅
    - 主に、並列探索処理などの用途

MPIの経緯 MPI-3.1

- MPI-3.0 2012年9月
- MPI-3.1 2015年6月
- 以下のページで現状・ドキュメントを公開中
  - http://mpi-forum.org/docs/docs.html
  - http://meetings.mpi-forum.org
  - http://meetings.mpi-forum.org/mpi31-impl-status-Nov15.pdf
- 注目すべき機能
  - ノン・ブロッキング集団通信機能（MPI_IALLREDUCE、など）
  - 高性能な片方向通信（RMA、Remote Memory Access)
  - Fortran2008 対応、など

MPIの経緯 MPI-4.0策定中

- 以下のページで経緯・ドキュメントを公開
  - https://www.mpi-forum.org/mpi-40/
- 検討されている機能
  - ハイブリッドプログラミングへの対応
    - Multiple Endpoint
  - MPIアプリケーションの耐故障性（Fault Tolerance, FT）
  - いくつかのアイデアを検討中
    - Active Messages (メッセージ通信のプロトコル)
      - 計算と通信のオーバラップ
      - 最低限の同期を用いた非同期通信
      - 低いオーバーヘッド、パイプライン転送
      - バッファリングなしで、インタラプトハンドラで動く
    - Stream Messaging
    - 新プロファイル・インターフェース

MPIの実装

- MPICH（エム・ピッチ）
  - 米国アルゴンヌ国立研究所が開発
- MVAPICH (エムヴァピッチ)
  - 米国オハイオ州立大学で開発、MPICHをベース
  - InfiniBand向けの優れた実装
- OpenMPI
  - オープンソース
- ベンダMPI
  - 大抵、上のどれかがベースになっている
    例: 富士通「京」、FX10用のMPI: Open-MPIベース
    Intel MPI, SGI MPE: MPICH、MVAPICHベース
  - 注意点:メーカ独自機能拡張がなされていることがある

MPIによる通信

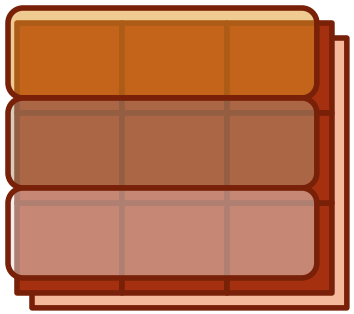
- 郵便物の郵送に同じ
- 郵送に必要な情報：
  1. 自分の住所、送り先の住所
  2. 中に入っているものはどこにあるか
  3. 中に入っているものの分類
  4. 中に入っているものの量
  5. （荷物を複数同時に送る場合の）認識方法（タグ）
- MPIでは：
  1. 自分の認識ID、および、送り先の認識ID
  2. データ格納先のアドレス
  3. データ型
  4. データ量
  5. タグ番号

略語とMPI用語

- MPIは「プロセス」間の通信を行います。プロセスは（普通は）「プロセッサ」（もしくは、コア）に一対一で割り当てられます。
- ランク（Rank）
  - 各「MPIプロセス」の「識別番号」のこと。
  - 通常MPIでは、MPI_Comm_rank関数で設定される変数に、0～全プロセス数-1　の数値が入る
  - コミュニケータ中のMPIプロセス数を知るために，MPI_Comm_size関数を使う。

コミュニケータ

- MPI_COMM_WORLDは、コミュニケータとよばれる概念を保存する変数
- コミュニケータは、操作を行う対象のプロセッサ群を定める
- 初期状態では、０番～numprocs－１番までのプロセッサが、１つのコミュニケータに割り当てられる
  - この名前が、“MPI_COMM_WORLD”
- プロセッサ群を分割したい場合、MPI_Comm_split 関数を利用
  - メッセージを、一部のプロセッサ群に放送するときに利用
  - “マルチキャスト”で利用
- 他にも様々な作成方法がある => 後述



MPIに含まれるもの

MPI3.1の目次に相当

- １対１通信関数
  - ブロッキング型
    - MPI_Send； MPI_Recv；
  - ノンブロッキング型
    - MPI_Isend； MPI_Irecv；
- 派生データ型
  - MPI_Type_
- 集団通信関数
  - MPI_Bcast；
    MPI_Reduce； MPI_Allreduce；
    MPI_Barrier；
- グループ、コミュニケータ
  - MPI_Comm_dup；MPI_Comm_split；
- プロセストポロジ
  - MPI_Cart_create；
- 環境の管理や表示
  - MPI_Init； MPI_Comm_rank；
    MPI_Comm_size； MPI_Finalize；
  - 時間計測関数
    - MPI_Wtime
- Infoオブジェクト
- プロセス生成・管理
- 片側通信
  - MPI_Put；MPI_Get；
- 外部インタフェース
- 並列ファイルIO（MPI-IO）
  - MPI_File_open,
- FortranとCのbinding
- ツールサポート

基本的なMPI関数

送信、受信のためのインタフェース

C言語インターフェースとFortranインターフェースの違い

- C版は、　整数変数ierr が戻り値

  ierr = MPI_Xxxx(….);

- Fortran版は、最後に整数変数ierrが引数

  call MPI_XXXX(…., ierr)

- システム用配列の確保の仕方
  - C言語

    MPI_Status istatus;
  - Fortran言語

    integer istatus(MPI_STATUS_SIZE)

C言語インターフェースとFortranインターフェースの違い

- MPIにおける、データ型の指定
  - C言語

    MPI_CHAR (文字型)、 MPI_INT (整数型)、MPI_FLOAT (実数型)、MPI_DOUBLE(倍精度実数型)
  - Fortran言語

    MPI_CHARACTER (文字型)、MPI_INTEGER (整数型)、MPI_REAL (実数型)、MPI_DOUBLE_PRECISION(倍精度実数型)、MPI_COMPLEX(複素数型)
- 以降は、Ｃ言語インタフェースで説明する

基礎的なMPI関数—MPI_Recv

- ierr = MPI_Recv(recvbuf, count, datatype, source, tag, comm, status);

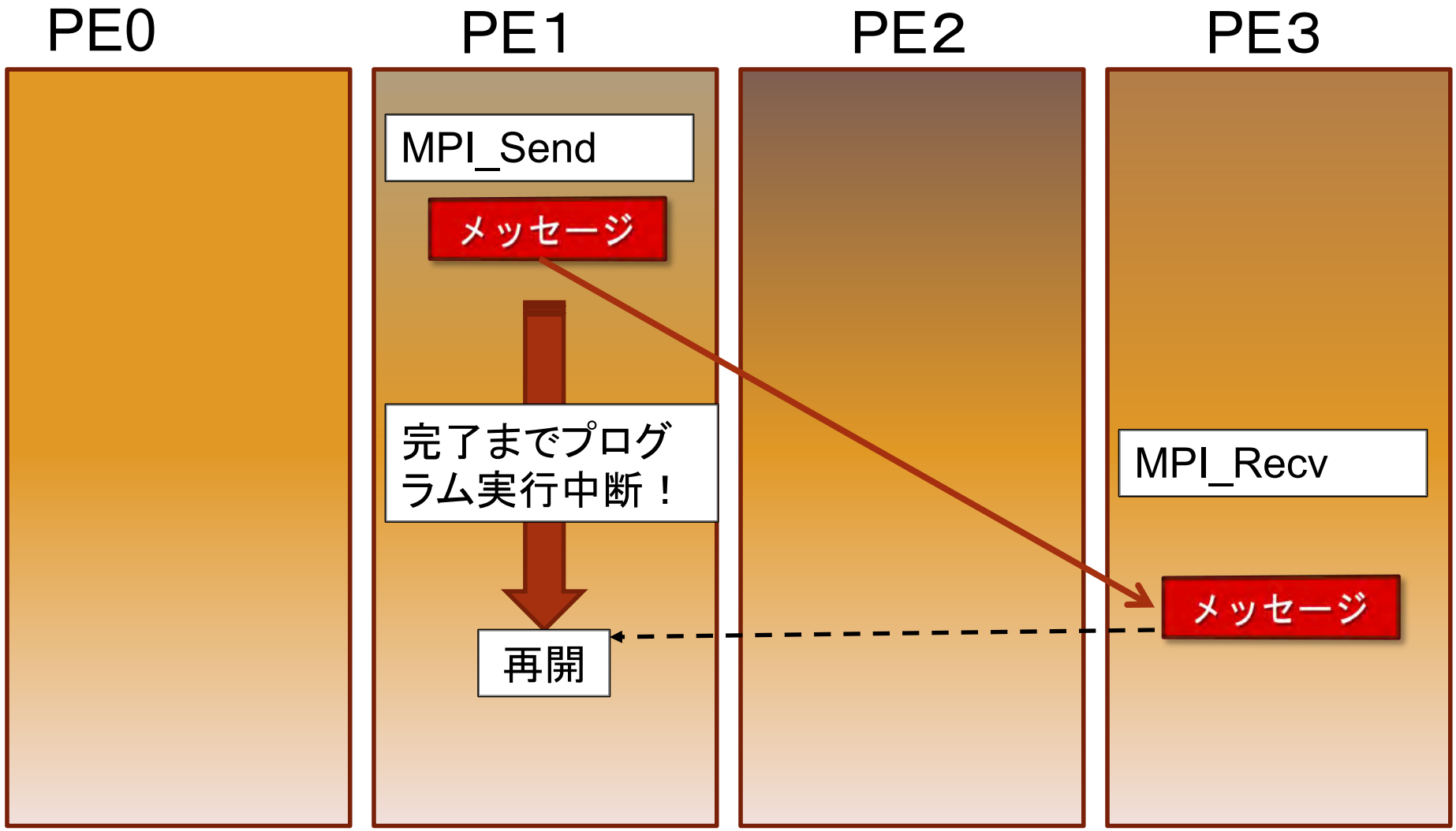
  - void * recvbuf (OUT): 受信領域の先頭番地を指定する。
  - int count (IN): 受信領域のデータ要素数を指定する。
  - int datatype (IN): 受信領域のデータの型を指定する。
    - MPI_CHAR (文字型)、MPI_INT (整数型)、MPI_FLOAT (実数型)、 MPI_DOUBLE(倍精度実数型)、など
  - int source (IN): 受信したいメッセージを送信するPEのランクを指定する。
    - 任意のPEから受信したいときは、MPI_ANY_SOURCE を指定
  - int tag (IN): 受信したいメッセージに付いているタグの値を指定。
    - 任意のタグ値のメッセージを受信したいときは、MPI_ANY_TAG を指定
  - MPI_Comm comm (IN): コミュニケータを指定。
    - 通常ではMPI_COMM_WORLD を指定すればよい。
  - MPI_Status status (OUT): 受信ステータス

基礎的なMPI関数—MPI_Send

- ierr = MPI_Send(sendbuf, count, datatype, dest, tag, comm, status);

  - void * sendbuf (IN): 送信領域の先頭番地を指定
  - int count (IN): 送信領域のデータ要素数を指定
  - int datatype (IN): 送信領域のデータの型を指定
  - int dest (IN): 送信したい相手のランクを指定
    - 任意のPEから受信したいときは、MPI_ANY_SOURCE を指定
  - int tag (IN): 受信したいメッセージに付いているタグの値を指定。
    - 任意のタグ値のメッセージを受信したいときは、MPI_ANY_TAG を指定
  - MPI_Comm comm (IN): コミュニケータを指定。
    - 通常ではMPI_COMM_WORLD を指定すればよい。

Send－Recvの概念（１対１通信）



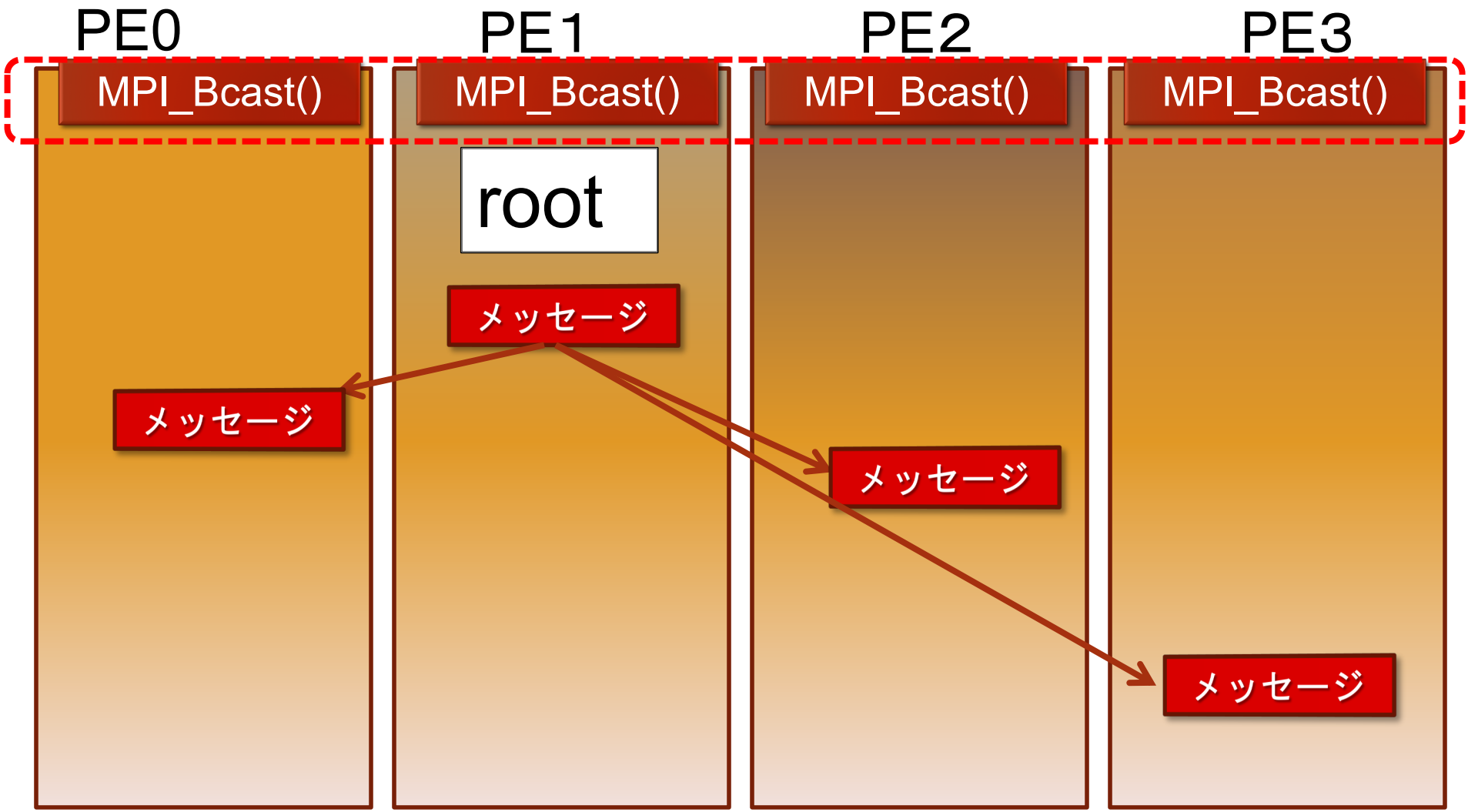
基礎的なMPI関数—MPI_Bcast

- ierr = MPI_Bcast(sendbuf, count, datatype,
  root, comm);

  - void * sendbuf (IN/OUT) : 送信(root)受信(root以外)領域の先頭番地
  - int count (IN) : 送信領域のデータ要素数
  - MPI_Datatype datatype (IN) : 送信領域のデータ型
  - int root (IN): 送信プロセスのランク番号
  - MPI_Comm comm (IN): コミュニケータ

全ランクが
同じように関数を呼ぶこと!!

MPI_Bcastの概念（集団通信）

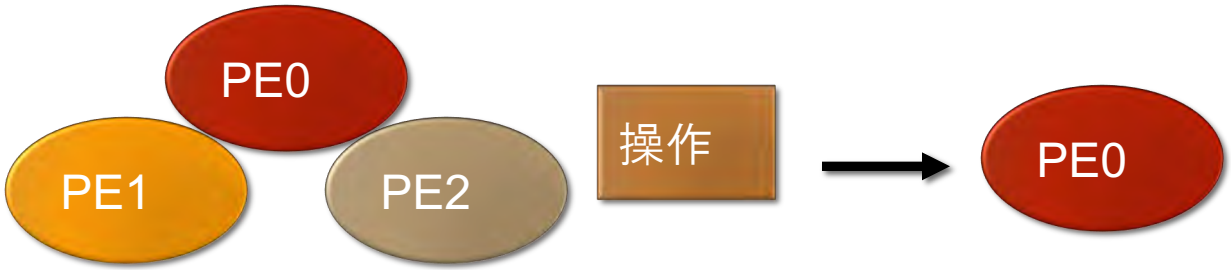


リダクション演算

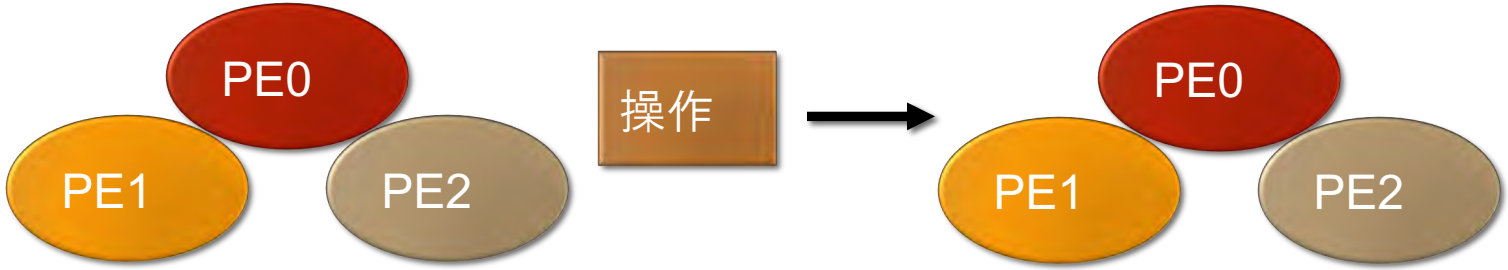
- ＜操作＞によって＜次元＞を減少（リダクション）させる処理
  - 例：　内積演算
    ベクトル（ｎ次元空間）　→　スカラ（１次元空間）
- リダクション演算は、通信と計算を必要とする
  - 集団通信演算（collective communication operation）
    と呼ばれる
- 演算結果の持ち方の違いで、２種のインタフェースが存在する

リダクション演算

- 演算結果に対する所有PEの違い
  - MPI_Reduce関数
    - リダクション演算の結果を、ある一つのPEに所有させる



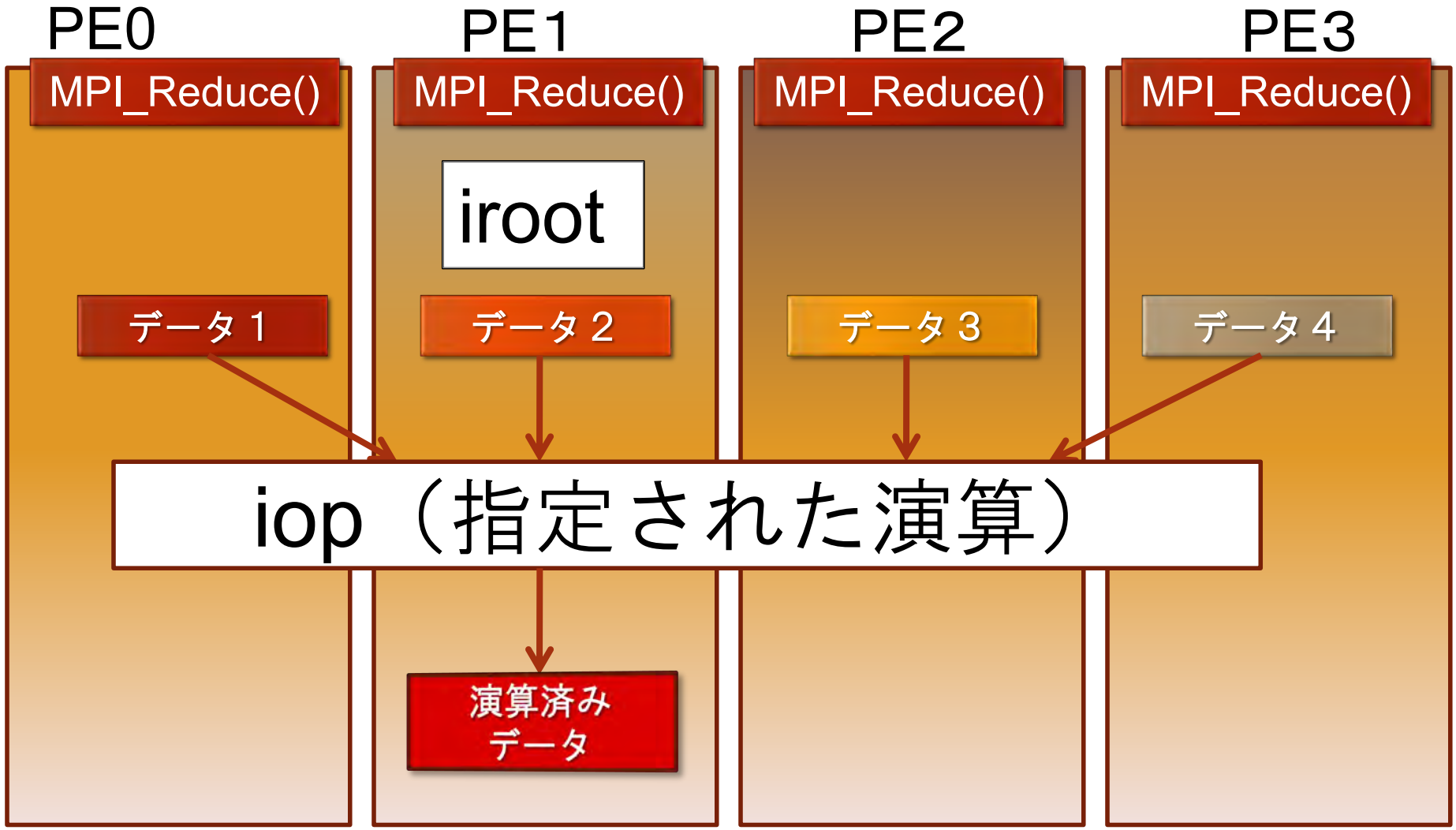
  - MPI_Allreduce関数
    - リダクション演算の結果を、全てのPEに所有させる



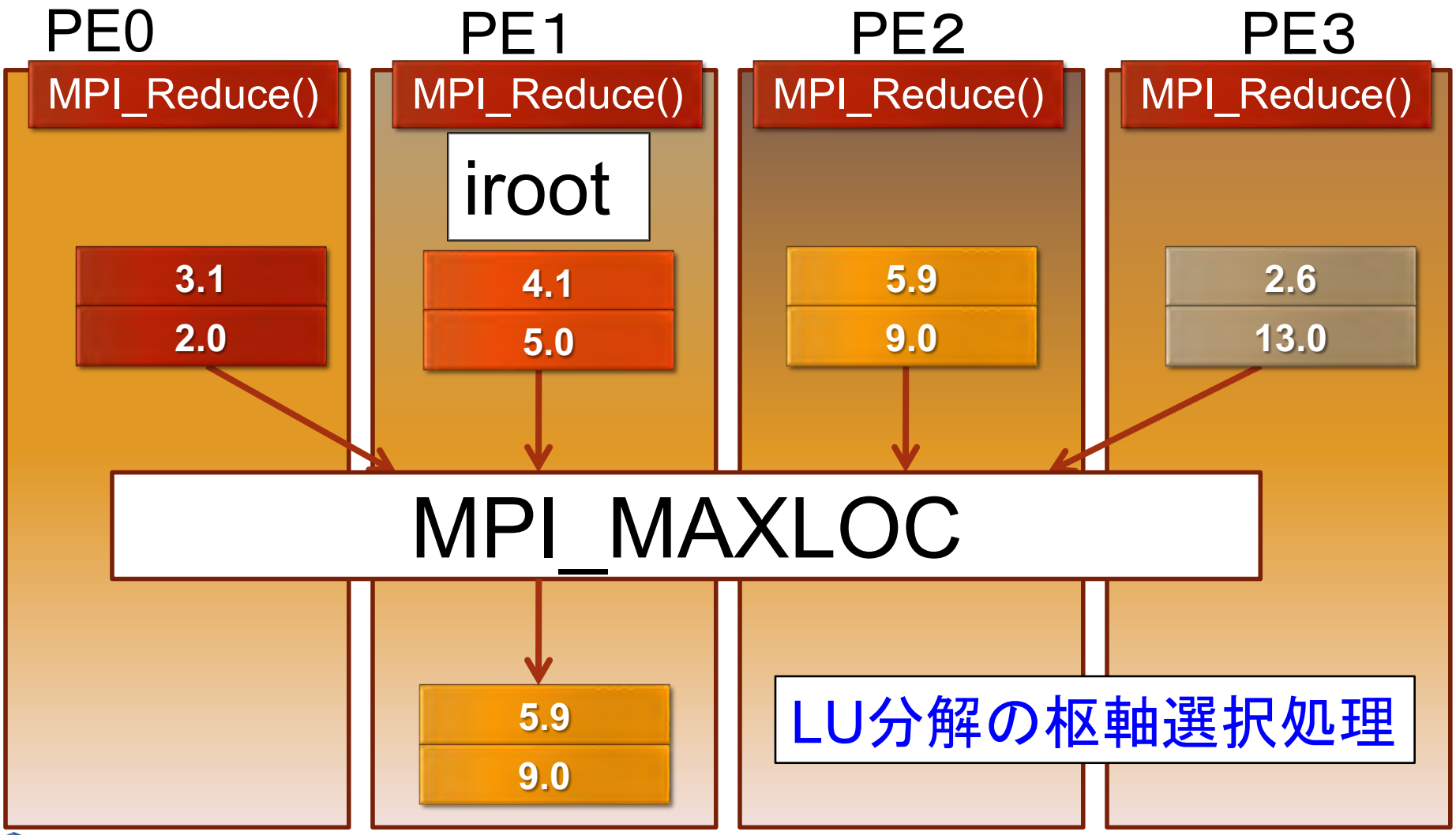
基礎的なMPI関数—MPI_Reduce

- ierr = MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm);
  - void * sendbuf (IN) : 送信領域の先頭番地
  - void * recvbuf (OUT) : 受信領域の先頭番地
    - 送信領域と受信領域は、同一であってはならない。
      すなわち、異なる配列を確保しなくてはならない。
  - int count (IN) : 送信領域のデータ要素数
  - MPI_Datatype datatype (IN) : 送信領域のデータ型
  - MPI_Op op (IN): 演算の種類
    - MPI_SUM (総和)、 MPI_PROD (積)、 MPI_MAX (最大)、MPI_MIN (最小)、 MPI_MAXLOC (最大とその位置)、 MPI_MINLOC (最小とその位置) など。
  - int root (IN): 送信プロセスのランク番号
  - MPI_Comm comm (IN): コミュニケータ

MPI_Reduceの概念（集団通信）



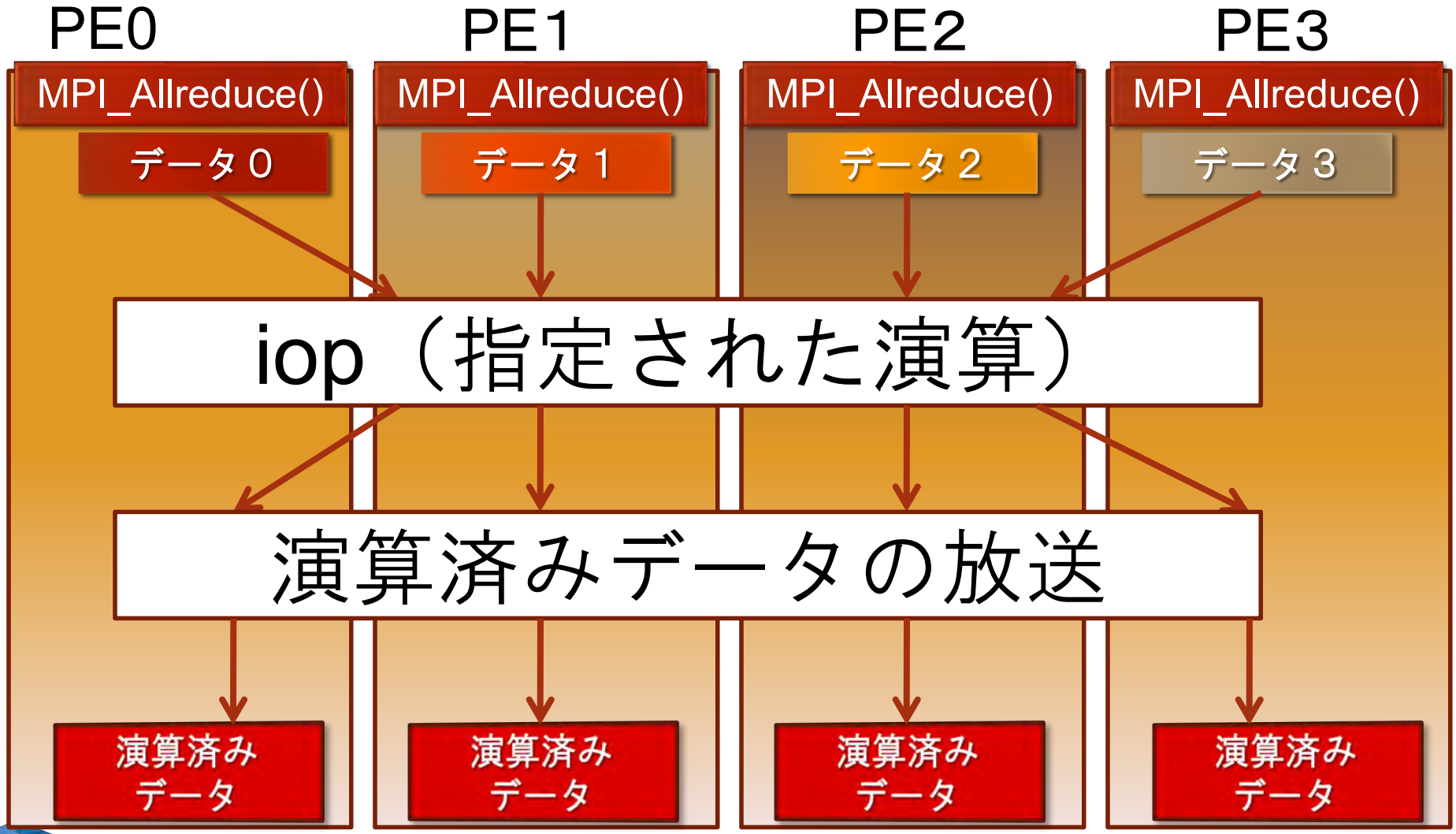
MPI_Reduceによる２リスト処理例
（MPI_2DOUBLE_PRECISION と MPI_MAXLOC）



基礎的なMPI関数—MPI_Allreduce

- ierr = MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm);
  - void * sendbuf (IN) : 送信領域の先頭番地
  - void * recvbuf (OUT) : 受信領域の先頭番地
    - 送信領域と受信領域は、同一であってはならない。
      すなわち、異なる配列を確保しなくてはならない。
  - int count (IN) : 送信領域のデータ要素数
  - MPI_Datatype datatype (IN) : 送信領域のデータ型
  - MPI_Op op (IN): 演算の種類
    - MPI_SUM (総和)、 MPI_PROD (積)、 MPI_MAX (最大)、MPI_MIN (最小)、 MPI_MAXLOC (最大とその位置)、 MPI_MINLOC (最小とその位置) など。
  - MPI_Comm comm (IN): コミュニケータ

MPI_Allreduceの概念（集団通信）



基礎的なMPI関数—MPI_Gather

- ierr = MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm);
  - void * sendbuf (IN) : 送信領域の先頭番地
  - int sendcount (IN) : 送信領域のデータ要素数
  - MPI_Datatype sendtype (IN) : 送信領域のデータ型
  - void * recvbuf (OUT) : 受信領域の先頭番地
    - 原則として、送信領域と受信領域は、同一であってはならない。すなわち、異なる配列を確保しなくてはならない。
  - int recvcount (IN): 受信領域のデータ要素数
  - MPI_Datatype recvtype (IN) : 受信領域のデータ型
    - root で指定したPEのみ有効 (recvbuf, recvcount, recvtype)
  - int root (IN): 受信プロセスのランク番号
  - MPI_Comm comm (IN): コミュニケータ

sendcount * size = recvcount

基礎的なMPI関数—MPI_Scatter

- ierr = MPI_Scatter (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm);
  - void * sendbuf (IN) : 送信領域の先頭番地
  - int sendcount (IN) : 送信領域のデータ要素数
  - MPI_Datatype sendtype (IN) : 送信領域のデータ型
    root で指定したPEのみ有効 (sendbuf, sendcount, sendtype)
  - void * recvbuf (OUT) : 受信領域の先頭番地
    - 原則として、送信領域と受信領域は、同一であってはならない。すなわち、異なる配列を確保しなくてはならない。
  - int recvcount (IN): 受信領域のデータ要素数
  - MPI_Datatype recvtype (IN) : 受信領域のデータ型
  - int root (IN): 送信プロセスのランク番号
  - MPI_Comm comm (IN): コミュニケータ

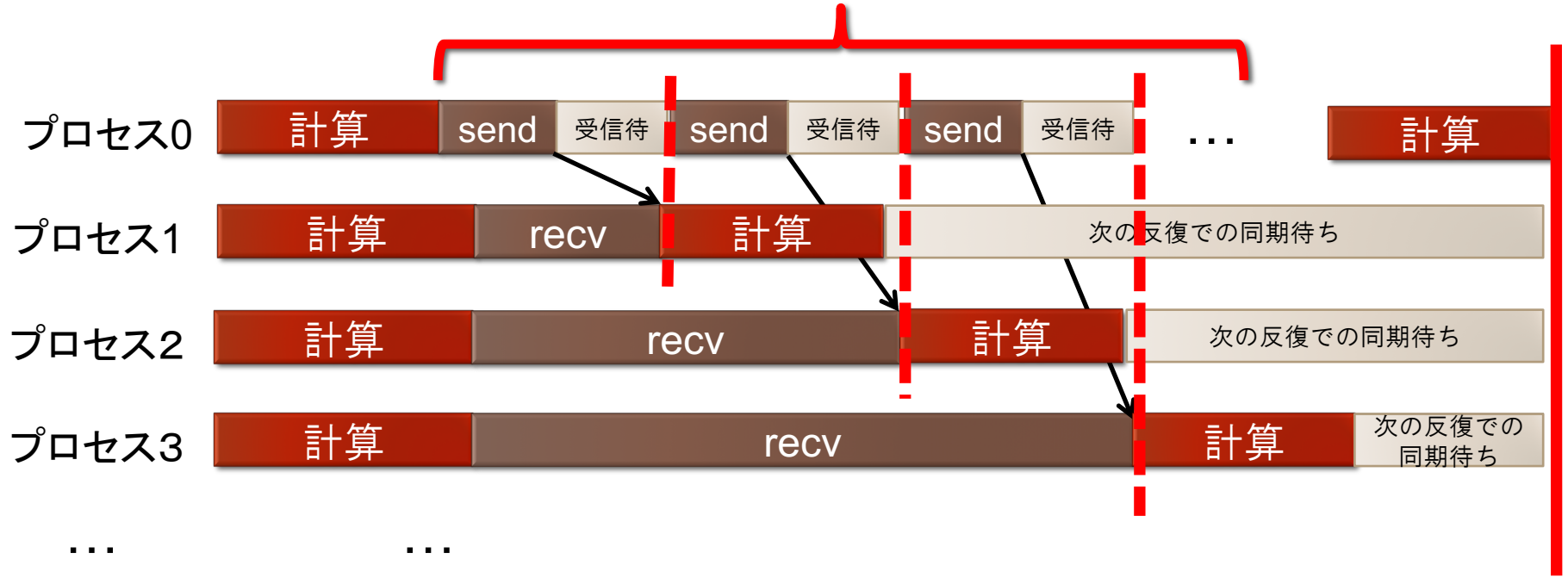
sendcount = recvcount * size

ブロッキング、ノンブロッキング

1. ブロッキング
   - 送信／受信側のバッファ領域にメッセージが格納され、受信／送信側のバッファ領域が自由にアクセス・上書きできるまで、呼び出しが戻らない
   - バッファ領域上のデータの一貫性を保障
   - MPI_Send, MPI_Bcastなど
2. ノンブロッキング
   - 送信／受信側のバッファ領域のデータを保障せずすぐに呼び出しが戻る
   - バッファ領域上のデータの一貫性を保障せず
     - 一貫性の保証はユーザの責任

ブロッキング通信で効率の悪い例

- プロセス0が必要なデータを持っている場合
  連続するsendで、効率の悪い受信待ち時間が多発



ノンブロッキング通信関数

- ierr = MPI_Isend(sendbuf, icount, datatype, idest, itag, icomm, irequest);

  - sendbuf：送信領域の先頭番地を指定する
  - icount：整数型。送信領域のデータ要素数を指定する
  - datatype：整数型。送信領域のデータの型を指定する
  - idest：整数型。送信したいPEのicomm 内でのランクを指定する
  - itag：整数型。受信したいメッセージに付けられたタグ
    の値を指定する

ノンブロッキング通信関数

- icomm： 整数型。PE集団を認識する番号であるコミュニケータを指定する。
  - 通常ではMPI_COMM_WORLD を指定すればよい。
- irequest： MPI_Request型（整数型の配列）。
  送信を要求したメッセージにつけられた識別子が戻る。
- ierr： 整数型。エラーコードが入る。

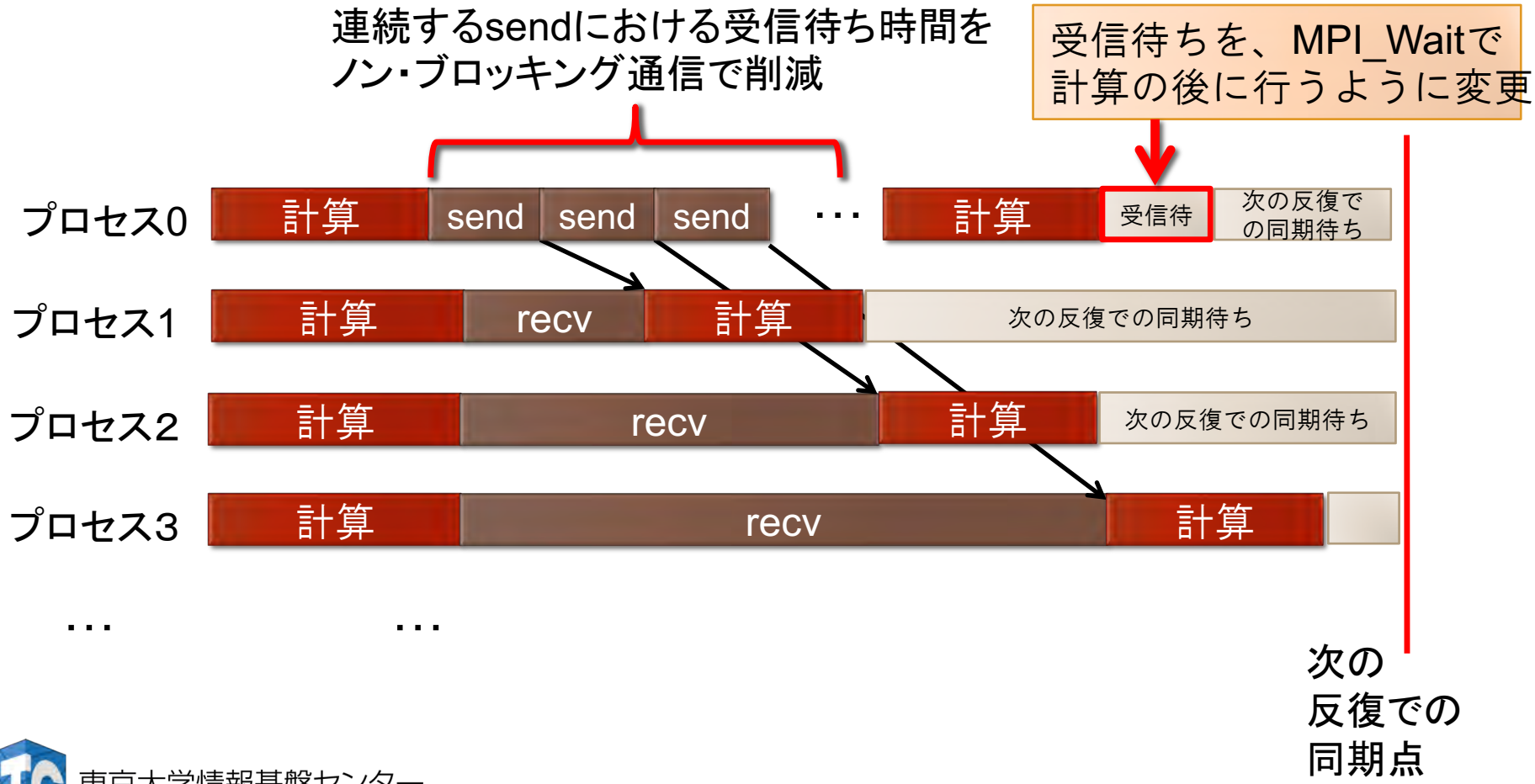
同期待ち関数

- ierr = MPI_Wait(irequest, istatus);

  - irequest： MPI_Request型（整数型配列）。
    送信を要求したメッセージにつけられた識別子。
  - istatus： MPI_Status型（整数型配列）。
    受信状況に関する情報が入る。
    - 要素数がMPI_STATUS_SIZEの整数配列を宣言して指定する。
    - 受信したメッセージの送信元のランクがistatus[MPI_SOURCE]、タグがistatus[MPI_TAG]に代入される。
  - 送信データを変更する前・受信データを読み出す前には必ず呼ぶこと

ノン・ブロッキング通信による改善

- プロセス0が必要なデータを持っている場合



注意点

- 以下のように解釈してください：
  - MPI_Send関数
    - 関数中にMPI_Wait関数が入っている；
  - MPI_Isend関数
    - 関数中にMPI_Wait関数が入っていない；
    - かつ、すぐにユーザプログラム戻る；

参考文献

1. Message Passing Interface Forum
   （ http://www.mpi-forum.org/ ）
2. MPI並列プログラミング、P.パチェコ 著 ／ 秋葉 博 訳
3. 並列プログラミング虎の巻MPI版、青山幸也 著、
   理化学研究所情報基盤センタ
   （ http://accc.riken.jp/HPC/training/text.html ）
4. MPI-Jメーリングリスト
   （ http://phase.hpcc.jp/phase/mpi-j/ml/ ）
5. 講習会資料ページ (RIST)
   http://www.hpci-office.jp/pages/seminar_text

Intel MPI

- Intel Parallel Studio XE cluster editionにて，C, Fortranコンパイラと一緒に提供
  - MPICH, MVAPICH2をベースに開発されているが，独自機能も多数入っている
- 最新版は2019
  - もうすぐインストールされるはず
- OFPで利用可能なバージョン
  - 2018 update 1 / 2 / 3
- 参考資料
  - Webポータル=>ドキュメント閲覧
  - https://software.intel.com/en-us/mpi-developer-guide-linux
  - https://software.intel.com/en-us/mpi-developer-reference-linux
  - https://software.intel.com/en-us/intel-software-technical-documentation?field_software_product_tid[]=20827

MVAPICH2

- オハイオ州立大で Prof. D.K. Pandaのグループが開発
  - Argonne National Lab.等で開発されているMPICHがベース (MPICH3)
- （主に）InfiniBand向け，最先端機能をいち早く実装
- 現時点での最新版： 2.3
- OFPで利用可能なバージョン: 2.3a
- 参考情報
  - http://mvapich.cse.ohio-state.edu
  - http://mvapich.cse.ohio-state.edu/static/media/mvapich/mvapich2-2.3-userguide.html

Open MPI

- LAM/MPIから発展
  - LAM: Ohio State Univ. => Univ. of Nortre Dam => Indiana Univ.
- 最新版は 3.1.2
  - 2.1.5
  - 3.0.2
  - 次は 4.0 の予定
- OFPではシステムにはインストールしていない
  - 各自でコンパイル、使用は可能
- 参考情報
  - https://www.open-mpi.org/doc/
    - はっきり言ってこれだけではよくわからない
  - https://www.open-mpi.org/faq/
    - MCAオプションの指定などはこちらを見ないと…

Oakforest-PACSで使用可能なMPI実装

- 他にも使えるものはあるが，今回使うのはこの3種

- Intel MPI 2018 Update1
  - OFPのデフォルト （2018/10現在）
  - Intelコンパイラ 2018u1とセット(intel/2018.1.163 impi/2018.1.163)
- intelIntel MPI 2018 Update 3
  - Intelコンパイラ 2018u3とセット(intel/2018.3.222 impi/2018.3.222)
  - module switch intel/2018.1.163 intel/2018.3.222
- MVAPICH2 2.3a
  - Intelコンパイラ 2018u1とセット (intel/2018.1.163 mvapich2/2.3a)
  - module switch impi mvapich2

参考：moduleコマンドの使い方

- 様々なコンパイラ，MPI環境などを切り替えるためのコマンド
- パスや環境変数など必要な設定が自動的に変更される
- ジョブ実行時にもコンパイル時と同じmoduleをloadすること

- 使用可能なモジュールの一覧を表示：module avail
- 使用中のモジュールを確認：module list
- モジュールのload: module load モジュール名
- モジュールのunload: module unload モジュール
- モジュールの切り替え：module switch 旧モジュール 新モジュール
- モジュールを全てクリア: module purge

パラメータの最適化

- MPI実装のデフォルトパラメータが最適とは限らない
  - 同じハードウェアで開発・調整しているわけではない
    - Oakforest-PACSのハードウェア(OPA)は特に最新の製品

プロトコルスタック選択

- export I_MPI_FABRICS_LIST={tmi,ofi}
  export I_MPI_FABRICS=ノード内:ノード間
  - shm: 共有メモリ（ノード内のみ選択可）
  - tmi: Intel2018.1 ではOmni Pathのおすすめ
  - ofi: 今後の標準、Intel2018.3以降
- 2018.1デフォルトでは shm:tmi 、経験上集団通信は tmi:tmi にした方が高速
- 2018.3デフォルトでは shm:ofi (tmiは使えなくなった）、集団通信ではofi:ofiがいい(?)

EagerとRendezvous

- Eagerプロトコル
  - 送信側はとにかく送信する
  - 受信側は
    - マッチするものがあれば受信処理
    - マッチしなければ，受信バッファに保持 => バッファへのコピーが発生
  - メッセージサイズ小のときに適している
- Rendezvousプロトコル
  - 送受信の前に，受信相手がいるかどうかを確認してから送る
=> ゼロコピーで通信が可能
  - メッセージサイズ大のときに適している

- 通常，メッセージサイズに応じて自動的に切り替わる
  - 切り替え点はユーザによっても指定可能
    - export I_MPI_EAGER_THRESHOLD=メッセージ長 (IntelMPI)
    - export MV2_IBA_EAGER_THRESHOLD=メッセージ長 (MVAPICH2)

アルゴリズムの選択

- 集団通信には複数のアルゴリズムが実装されている
- 例：MPI_Allreduce
  1. Recursive doubling
  2. Rabenseifner’s
  3. Reduce + Bcast
  4. Topology aware Reduce + Bcast
  5. Binomial gather + scatter
  6. Topology aware binominal gather + scatter
  7. Shumilin's ring
  8. Ring
  9. Knomial
  10. Topology aware SHM-based flat
  11. Topology aware SHM-based Knomial
  12. Topology aware SHM-based Knary
- export I_MPI_ADJUST_ALLREDUCE=8:1024-4096@16-32;…
  - 1024~4096 Byteかつ16~32ノードのときRingアルゴリズムを使う

ノンブロッキング集団通信

- MPI-3.0で定義
- 集団通信のノンブロッキング版，MPI_Ixxx

分類	ブロッキング	ノンブロッキング	MPI_IN_PLACE	Intercomm
One-to-All	MPI_Bcast	MPI_Ibcast		✓
	MPI_Scatter{,v}	MPI_Iscatter{,v}	recvbuf@root	✓
All-to-One	MPI_Gather{,v}	MPI_Igather{,v}	sendbuf@root	✓
	MPI_Reduce	MPI_Ireduce	sendbuf@root	✓
All-to-All	MPI_Allgather{,v}	MPI_Iallgather{,v}	sendbuf	✓
	MPI_Alltoall{,v,w}	MPI_Ialltoall{,v,w}	sendbuf	✓
	MPI_Allreduce	MPI_Iallreduce	sendbuf	✓
	MPI_Reduce_scatter{,_block}	MPI_Ireduce_scatter{,_block}	sendbuf	✓
	MPI_Barrier	MPI_Ibarrier		✓
Others	MPI_Scan, Exscan	MPI_Iscan, Iexscan	sendbuf	

特別な変数

- 送受信で同じ配列を指定したい場合
  - collective通信で単に同じ配列をsendbuf, recvbufに指定するとエラーになる
    => 代わりにMPI_IN_PLACEを使う
    - 関数によって，sendbuf側かrecvbuf側に指定：前ページの表を参照

- 変数の指定を省略したい場合
  - MPI_STATUS_IGNORE
    - MPI_Recvなどで，MPI_Status の返り値が不要な場合
  - MPI_STATUSES_IGNORE
    - MPI_Waitallなどで MPI_Status []の返り値が不要な場合
  - MPI_ERRCODES_IGNORE
    - Fortranでierrの返り値が不要な場合

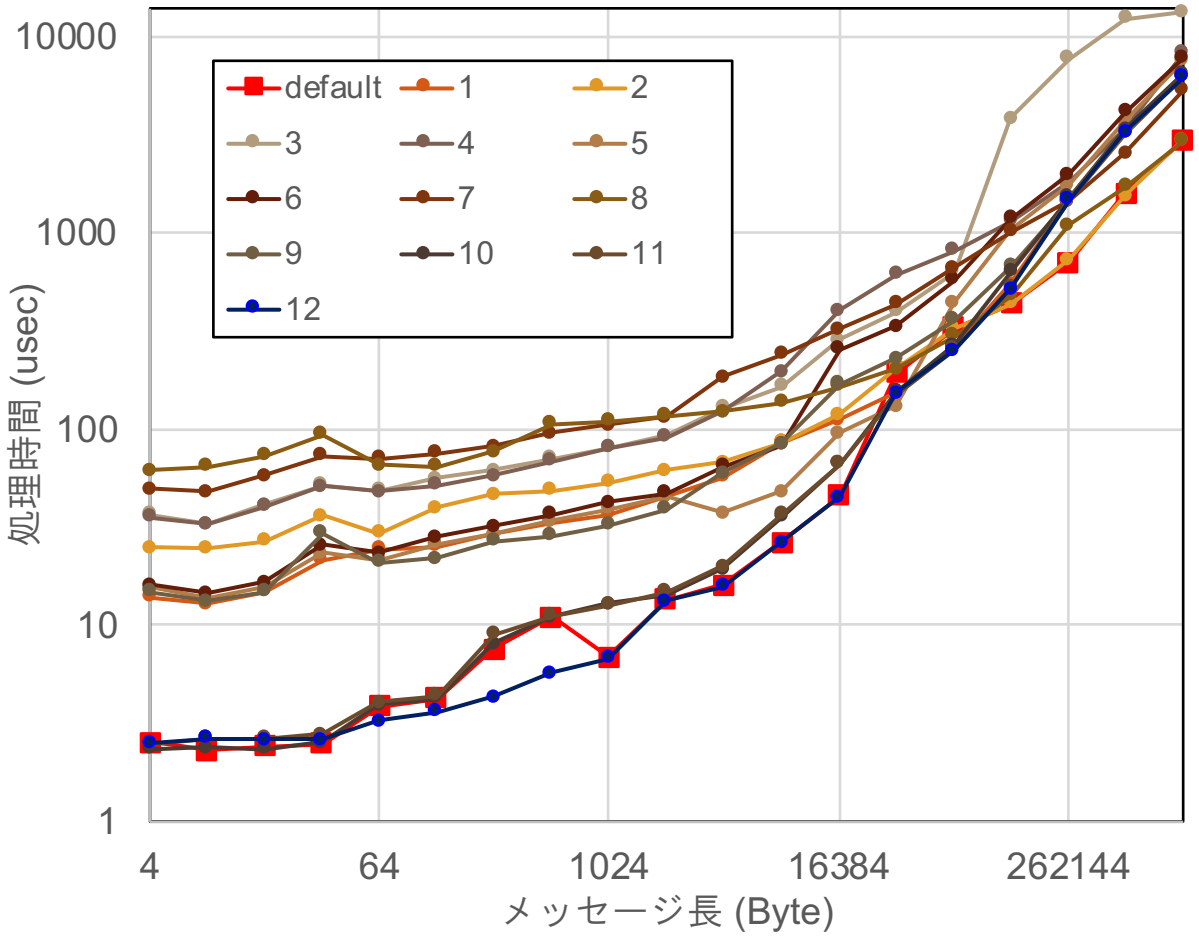
演習

- 各種MPI実装の性能を確認してみよう

Allreduceのアルゴリズムによる性能差

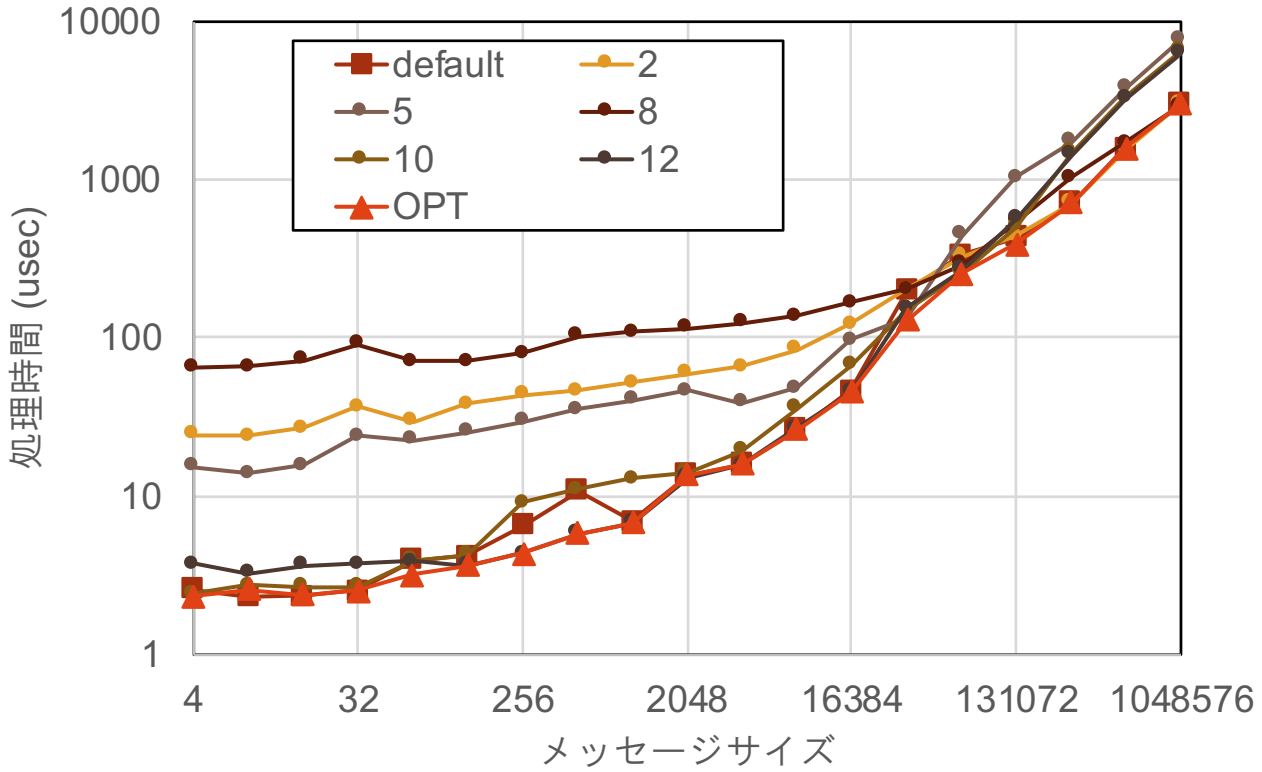
- 16ノード
- 1024プロセス
- Intel MPI
- tmi:tmi

- サイズによって最適アルゴリズムが変わる



Allreduceの性能改善 (16ノードx64の場合)

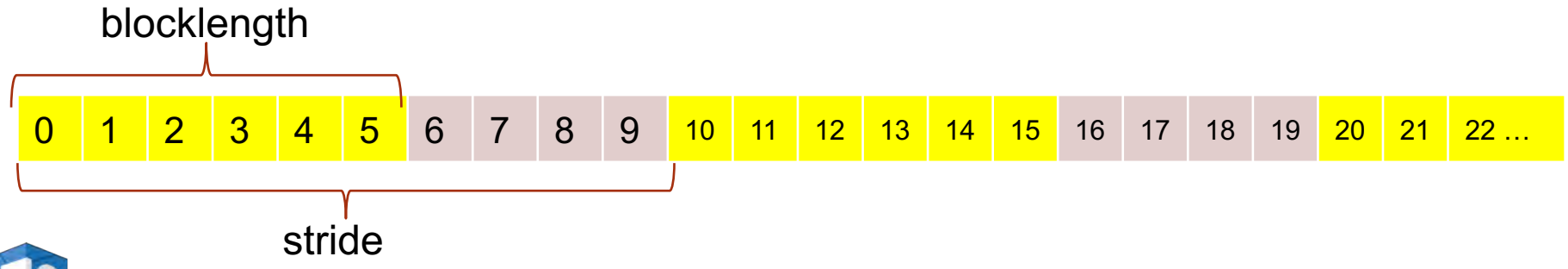
- メッセージ長ごとに最適なアルゴリズムを組み合わせ
  - export I_MPI_ADJUST_ALLREDUCE="10:1-32;12:33-16384;5:16385-32768;10:32769-65536;2:65537-524288;8:524289-1048576"



派生データ型：新しいデータ型の定義

ブロックストライド転送（同一型，規則的）

- ierr = MPI_Type_vector(count, blocklength, stride, oldtype, newtype)
  - int count (IN): 繰り返し回数（strideの）
  - int blocklength (IN): ブロック数（連続データの個数）
  - int stride (IN): ブロック開始の間隔
  - MPI_Type oldtype (IN): 元のデータ型
  - MPI_Type * newtype (OUT):新しいデータ型
- 利用する前に MPI_Type_commitを忘れずに!!



同一型，不規則なデータ配置

- ierr = MPI_Type_indexed(count, array_of_blocklengths,
  array_of_displacements, oldtype, newtype);

- int count (IN): ブロックの数
- int array_of_blocklengths[] (IN): 各ブロックの要素数
- int array_of_displacements[] (IN): 次のブロックまでの間隔
- MPI_Datatype oldtype (IN): 元のデータ型
- MPI_Datatype *newtype (OUT): 新しいデータ型

同一型，不規則なデータ配置(2)

- ierr = MPI_Type_create_hindexed(count, array_of_blocklengths, array_of_displacements, oldtype, newtype);

  - int count (IN): ブロックの数
  - int array_of_blocklengths[] (IN): 各ブロックの要素数
  - MPI_Aint array_of_displacements[] (IN): 次のブロックまでの間隔（アドレス値）
  - MPI_Datatype oldtype (IN): 元のデータ型
  - MPI_Datatype *newtype (OUT): 新しいデータ型

- MPI_Type_indexedとはarray_of_displacementのみが異なる
- MPI-2.0以前はMPI_Type_hindexed

異なる型，不規則なデータ配置

- ierr = MPI_Type_create_struct(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype);

  - int count (IN): メンバの数
  - int array_of_blocklengths[] (IN): 各メンバの個数
  - MPI_Aint array_of_displacements[] (IN):次のメンバまでの間隔（アドレス値）
  - MPI_Datatype array_of_types[] (IN): 各メンバのデータ型
  - MPI_Datatype *newtype (OUT): 新しいデータ型

- MPI-2.0以前はMPI_Type_struct

ファイルシステムとMPI-IO

Oakforest-PACSで利用可能なファイルシステム

PATH	種類	備考
/home/ログイン名	NFS	ログインノードからのみ利用可能 容量が小さい ログインに必要なもの・各種設定ファイルなど、最低限のものだけ置くこと
/work/グループ名/ログイン名	並列(Lustre)	ログインノードからも計算ノードからも利用可能，一般的な用途に使える バッチジョブの投入はここから行う
/cache/グループ名/ログイン名 ime://work/グループ名/ログイン名 （MPI-IO)	バーストバッファ(IME)	Lustreのキャッシュのように動作 ログインノードからは直接参照できないのでステージングが必要
/tmp	Ramdisk	非常に容量が小さい 使用を推奨しない
/dev/shm		

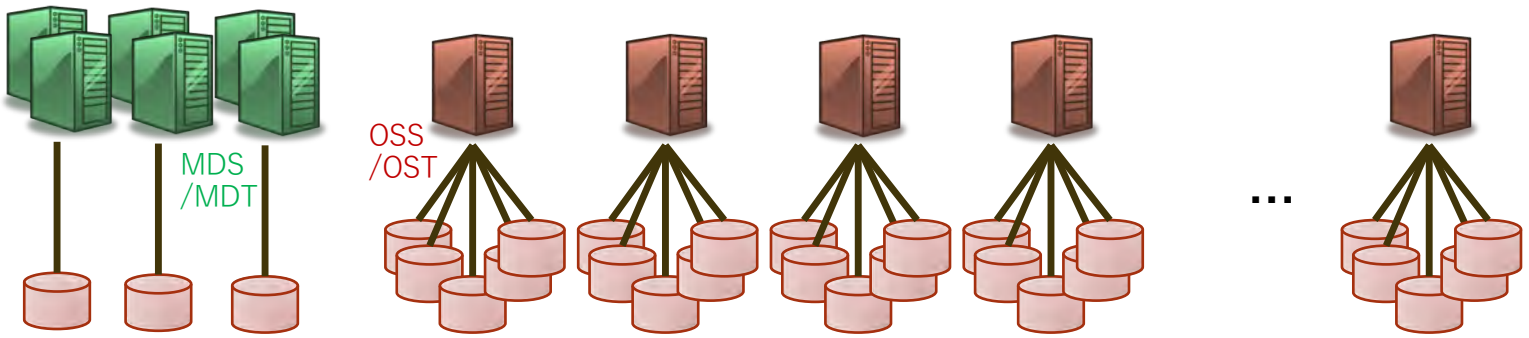
Oakforest-PACSの並列ファイルシステム

- Lustreファイルシステム
  - 大規模ファイル入出力、メタデータ操作の両方で高性能なファイルシステム
  - データの分散方法をファイルごとに指定可能(後述)
  - オープンソース開発されており様々なベンダーが採用している
  - 富士通FEFS: Lustreファイルシステムの上位互換, Reedbush

Lustreの物理構成とデータ配置

- メタデータを格納するMDS/MDT (Meta Data Server/Target)
  - ファイル管理情報（日付、サイズ等）、OSS/OSTへのデータ格納情報
- データを格納するOSS/OST (Object Storage Server/Target)
  - ファイルデータそのもの
- OFPの構成
  - MDT：3 領域 (DNE: Distributed Namespace)
  - OSS：「RAID6 (8D2P) × 41 + 10 スペア」 を10セット => 26 PB

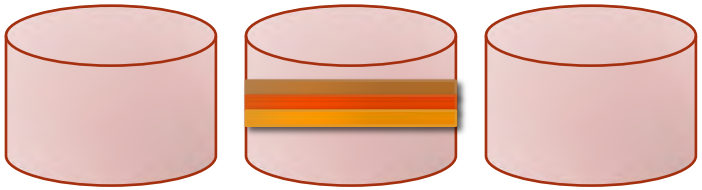
D: データ
P: パリティ



Lustreのデータ配置の指定

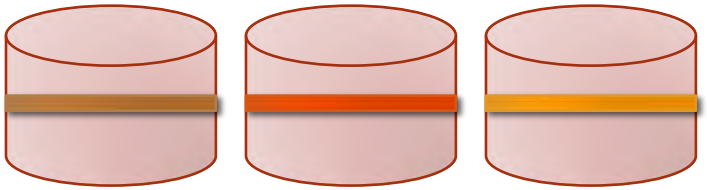
- データ配置の指定
  - １ファイルのデータをひとつのOSTに配置するか、複数のOSTに分散して配置するかはユーザが指定できる
    - デフォルトでは１ファイルあたりひとつのOSTに配置、ファイル単位で使用するOSTが決められる
    - lfs getstripe / lfs setstripeコマンドで参照・変更可能
  - 複数プロセスから単一のファイルに対する処理を高速化したい場合には複数のOSTを使うような指定が必要

ひとつのOSTに配置



どんなにがんばっても最大で1OST分の読み書き性能しか得られない

複数のOSTに配置



複数OST分の読み書き性能が得られる可能性がある

Lustreのデータ配置の指定の方法

- lfs setstripe [OPTIONS] <ディレクトリ名|ファイル名>
  - 主なオプション：-s size -c count
    - ストライプサイズ size 毎にcount 個のOSTに渡ってデータを分散配置する、という設定にした空のファイルを作成する
    - 既存ディレクトリに対して行うとその後に作るファイルに適用される
    - countに-1を指定すると全OSTを使用
  - 使用例

※ファイルを作成する際に指定する必要があるため、最初に削除している

```
$ rm /path/to/data.dat
$ lfs setstripe -s 1M -c 50 /path/to/data.dat
$ dd if=/dev/zero of=/path/to/data.dat bs=1M count=4096
```

- lfs getstripe <ディレクトリ名|ファイル名>
  - 設定情報を確認する
- lfs df
  - MDT/OST構成情報を確認する

バーストバッファ (DDN: Infinite Memory Engine)

- SSD搭載のサーバが25台，計50ノード
- クライアントでパリティを計算して書き込む
  - デフォルト： 9D+1P での動作
  - ユーザ指定可能
    export IM_CLIENT_PGEOM=<D>+<P>
  - 全てをデータ，パリティ不要であれば
    export IM_CLIENT_PGEOM=10+0

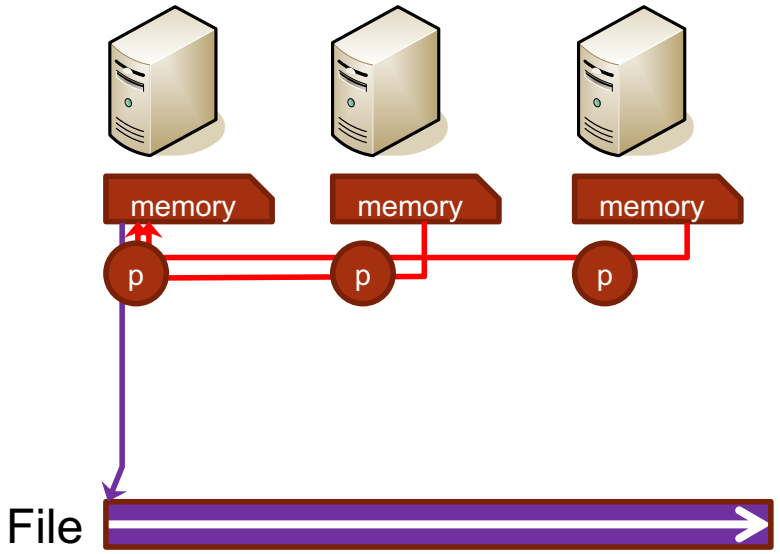
MPI-IO とは

- 並列ファイルシステムをMPIの枠組みで効率的に利用するための仕組み
  - ある程度抽象化を持たせた記述をすることで、（利用者が気にすることなく）最適な実装が利用可能になる（ことが期待される）
  - 例：利用者はファイル上のデータが配列上のどこに配置されて欲しいかだけを指定
    → MPI-IOにより「まとめて読み込んでMPI通信で分配配置」されて高速に処理が行われる、かも知れない

- 以後 API は C言語での宣言や利用例を説明するが、Fortranでも同名の関数が利用できる
  - 具体的な引数の違いなどはリファレンスを参照のこと
  - C++宣言はもう使われていない、C宣言を利用する

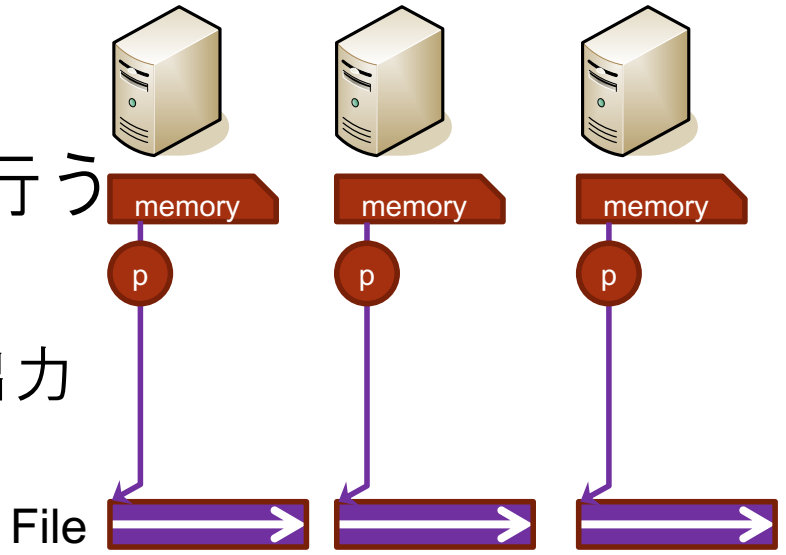
非MPI-IO：並列アプリによるファイルI/Oの例1

- 逐次入出力
  - 1プロセスのみがI/Oを行う、(MPI)通信によりデータを分散・集約する
    - MPI_Gather + fwrite
    - fread + MPI_Scatter
  - 利点
    - 単一ファイルによる優秀な取り回し
    - 読み書き操作回数の削減
  - 欠点
    - スケーラビリティの制限
    - 並列入出力をサポートしたファイルシステムの性能を活かせない



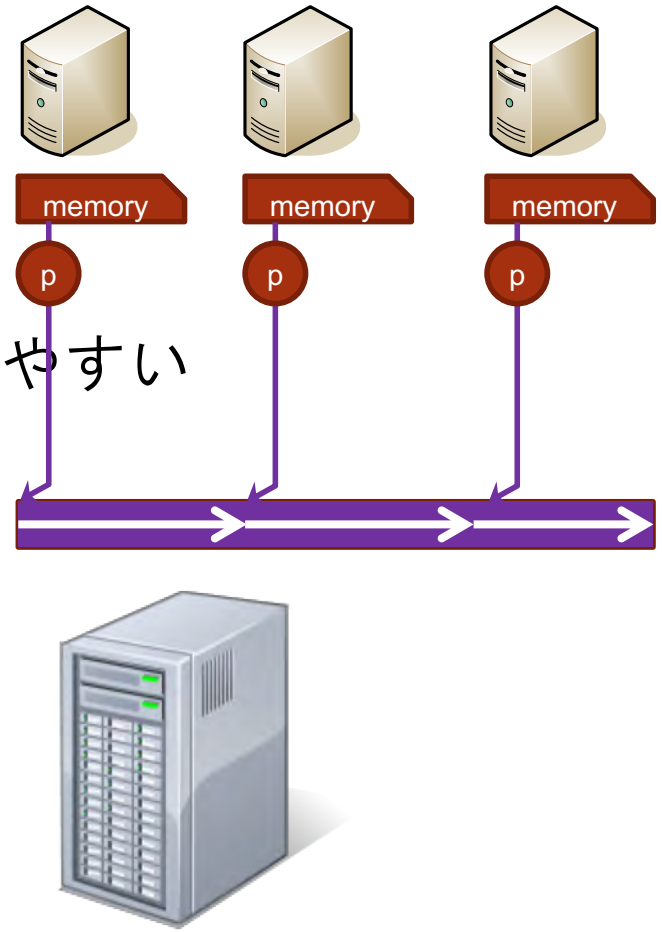
非MPI-IO：並列アプリによるファイルI/Oの例2

- 並列入出力
  - 各プロセスが個別にI/Oを行う
  - 利点
    - ファイルシステムの並列入出力サポートを生かせる
    - スケーラビリティ向上
  - 欠点
    - 入出力回数の増大
      - 多数の小ファイルアクセス
    - 複数ファイルによる劣悪な取り回
      - プログラム（ソース）も読み/書きにくい



MPI-IOによる並列アプリ上ファイルI/Oの一例

- 単一ファイルに対する並列入出力
  - スケーラビリティ向上
  - プログラム（ソース）もわかりやすい



想定するシナリオ

- MPI-IO関数を使うことで、複数プロセスによる単一ファイルへの並列入出力を簡単に行うことを考える
  - 今回は利用方法の習得と利便性の理解を目的とし、入出力性能についてはこだわらない
  - 単一ファイルに対する操作のため、入出力の性能も上げたい場合にはlfs setstripeの設定も必要

MPI-IOによる並列ファイル入出力

- 基本的な処理の流れ

1. MPI_File_open 関数によりファイルをオープン
2. 読み書きの処理を実行（様々な関数が用意されている）
3. MPI_File_close 関数によりファイルをクローズ

- ファイルのオープンとクローズ

```
int MPI_File_open(
  MPI_Comm comm,   // コミュニケータ
  char *filename,  // 操作ファイル名
  int amode,       // アクセスモード（読み書き、作成など）
  MPI_Info info,   // 実装へのユーザからのヒント
  MPI_File *fh     // ファイルハンドラ
)

int MPI_File_close(
  MPI_File *fh     // ファイルハンドラ
)
```

読み書きする場所・パターンの指定

- いくつかの指定方法がある
  - _at 系のread/write関数でその都度指定する
  - MPI_File_seek であらかじめ指定しておく

```
int MPI_File_seek(
  MPI_File fh,        // ファイルハンドラ
  MPI_Offset offset,  // オフセット（バイト数）
  int whence          // 指定方法のバリエーション ※
)
```

※即値、現在位置からのオフセット、ファイル末尾からのオフセット

  - MPI_File_set_view であらかじめ指定しておく

```
int MPI_File_set_view(
  MPI_File fh,          // ファイルハンドラ
  MPI_Offset disp,      // オフセット（バイト数）
  MPI_Datatype dtype,   // 要素データ型
  MPI_Datatype filetype, // ファイル型 ※
  char* datarep,        // データ表現形式
  MPI_Info info         // 実装へのユーザからのヒント
)
```

※要素データ型と同じ基本型または要素データ型で構成される派生型

読み書き方法のバリエーション

- 「view」を用いた書き込みだけでも様々なバリエーションが存在
  - ブロッキング・非集団出力
    - MPI_File_write
  - 非ブロッキング・非集団出力
    - MPI_File_iwrite / MPI_Wait
  - ブロッキング・集団出力
    - MPI_File_write_all
  - 非ブロッキング・集団出力
    - MPI_File_write_all_begin / MPI_File_write_all_end
- 用途に合わせて使い分ける
  - 非ブロッキング：読み書きが終わらなくても次の処理を実行できる
  - 集団出力：同一コミュニケータの全プロセスが行わねばならない、まとめて行われるため読み書き処理の時間自体は高速

並列出力の例：MPI_File_set_view/writeの場合

- プロセス番号（MPIランク）に基づいて1つずつ整数を書き出すだけの単純な例

```
MPI_File mfh;
MPI_Status st;
int disp;
int data;
const char filename[] = "data1.dat";

MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

disp = sizeof(int)*rank; // 各プロセスが書き込む場所を設定
data = 1+rank; // 書き込みたいデータを設定（MPIランク+1）

MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_RDWR|MPI_MODE_CREATE,
MPI_INFO_NULL, &mfh);

MPI_File_set_view(mfh, disp, MPI_INTEGER, MPI_INTEGER, "native", MPI_INFO_NULL);
MPI_File_write(mfh, &data, 1, MPI_INTEGER, &st);
MPI_File_close(&mfh);

MPI_Finalize();
```

出力結果の例（OFP 16プロセス）

```
$hexdump data1.dat
0000000 0001 0000 0002 0000 0003 0000 0004 0000
0000010 0005 0000 0006 0000 0007 0000 0008 0000
0000020 0009 0000 000a 0000 000b 0000 000c 0000
0000030 000d 0000 000e 0000 000f 0000 0010 0000
0000040
```

利点

- データの集約処理は不要
- 書き込み結果が1ファイルにまとまる

並列出力の例：その他の関数の場合

- MPI_File_iwrite / MPI_Wait

```
MPI_File_set_view(mfh, disp, MPI_INTEGER, MPI_INTEGER, "native",
MPI_INFO_NULL);
MPI_File_iwrite(mfh, &data, 1, MPI_INTEGER, &req);
MPI_Wait(&req, &st);
```

- MPI_File_write_all

```
MPI_File_set_view(mfh, disp, MPI_INTEGER, MPI_INTEGER, "native",
MPI_INFO_NULL);
MPI_File_write_all(mfh, &data, 1, MPI_INTEGER, &st);
```

- MPI_File_write_all_begin / MPI_File_write_all_end

```
MPI_File_set_view(mfh, disp, MPI_INTEGER, MPI_INTEGER, "native",
MPI_INFO_NULL);
MPI_File_write_all_begin(mfh, &data, 1, MPI_INTEGER);
MPI_File_write_all_end(mfh, &data, &st);
```

- MPI_File_write_at

```
MPI_File_write_at(mfh, disp, &data, 1, MPI_INTEGER, &st);
```

いずれも前ページのMPI_File_openとMPI_File_closeの間を置き換えて使う

互換性・可搬性について

- MPI_File_set_viewのdatarep引数によるデータ表現形式によって可搬性を向上させることができる
  - “native”: メモリ上と同じ姿での表現（何も変換しない）
  - “internal”: 同じMPI実装を利用するとき齟齬がない程度の変換
  - “external32”: MPIを利用する限り齟齬がないように変換
  - その他のオプションはMPI仕様書を参照
- 「View」を使わない入出力処理は可搬性が保証されない
  - MPI_File_open → MPI_File_wite_at → MPI_File_close は記述量が少なく簡単だが、可搬性が低い
- 具体的な例
  - Oakforest-PACSと京コンピュータで同じプログラム（ソースコード）を使いたい
    - nativeでは京コンピュータとOakforest-PACSの出力結果が一致しない
    - external32では出力結果が一致する（京コンピュータの結果に揃う）
  - Oakforest-PACS同士でもMPI処理系を変更すると結果が変わるかも知れない

共有ファイルポインタ

- ファイルポインタを共有することもできる
  - MPI_File_read/write_sharedで共有ファイルポインタを用いて入出力
  - 読み書き動作が他プロセスの読み書きにも影響する、「カーソル」の位置が共有される
  - 複数プロセスで順番に（到着順に）１ファイルを読み書きするような際に使う
    - 例：ログファイルへの追記
- 共有ファイルポインタを用いた集団入出力もある
  - MPI_File_read/write_ordered
  - 順序が保証される
    - ランク順に処理される
    - 並列ではない

```
#define COUNT  2
MPI_File fh;
MPI_Status st;
int buf[COUNT];
MPI_File_open
  (MPI_COMM_WORLD, “datafile”,
  MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_read_shared
  (fh, buf, COUNT, MPI_INT, &st);
MPI_File_close(&fh);
```

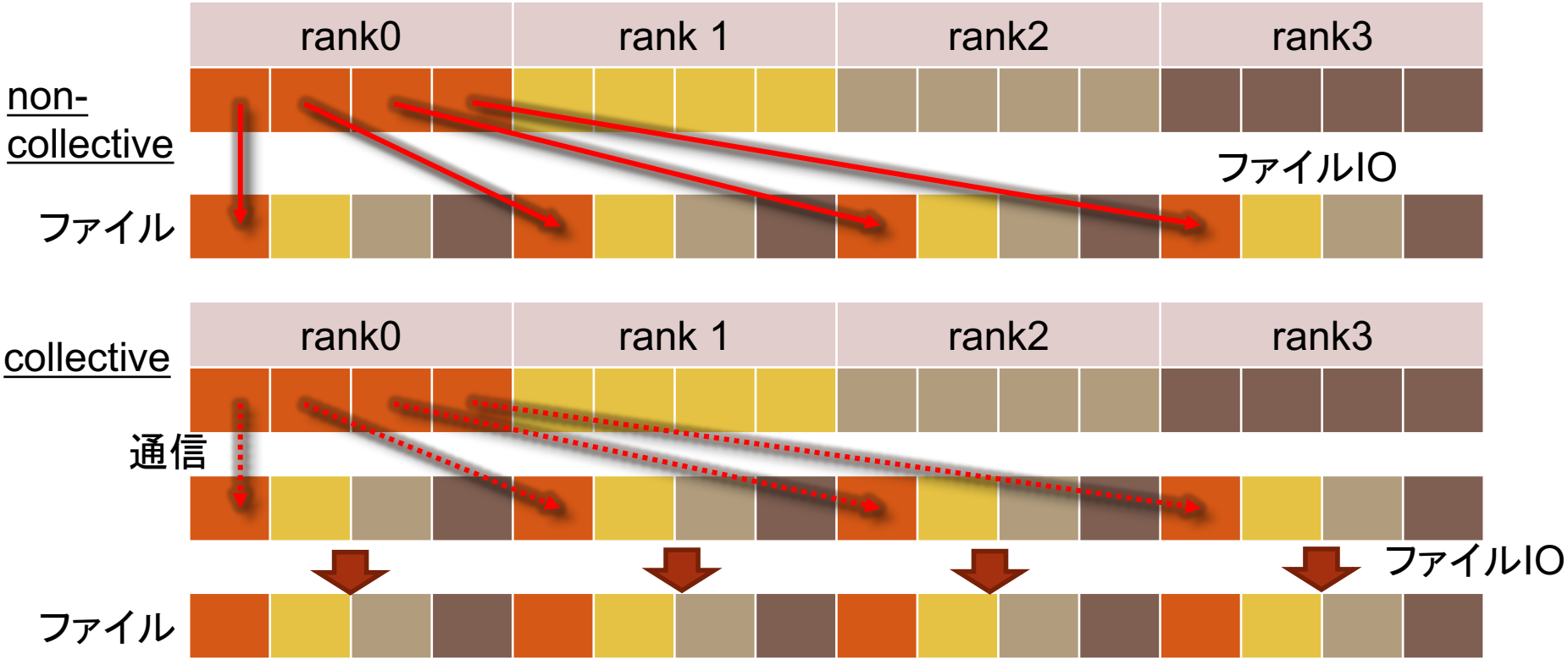
ファイル入出力関数のまとめ

位置	同期	non-collective	collective
明示的オフセット	blocking	MPI_FILE_{READ,WRITE}_AT	MPI_FILE_{READ,WRITE}_AT_ALL
	nonblocking	MPI_FILE_I{READ,WRITE}_AT	MPI_FILE_I{READ,WRITE}_AT_ALL
	split collective	N/A	MPI_FILE_{READ,WRITE}_AT_ALL_{BEGIN,END}
個別ファイルポインタ	blocking	MPI_FILE_{READ,WRITE}	MPI_FILE_{READ,WRITE}_ALL
	nonblocking	MPI_FILE_I{READ,WRITE}	MPI_FILE_I{READ,WRITE}_ALL
	split collective	N/A	MPI_FILE_{READ,WRITE}_ALL_{BEGIN,END}
共有ファイルポインタ	blocking	MPI_FILE_{READ,WRITE}_SHARED	MPI_FILE_{READ,WRITE}_ORDERED
	nonblocking	MPI_FILE_I{READ,WRITE}_SHARED	N/A
	split collective	N/A	MPI_FILE_{READE,WRITE}_ORDERED_{BEGIN,END}

OFPでは MPI_FILE_WRITE_SHARED はサポートしない（他にも共有ポインタは要確認）

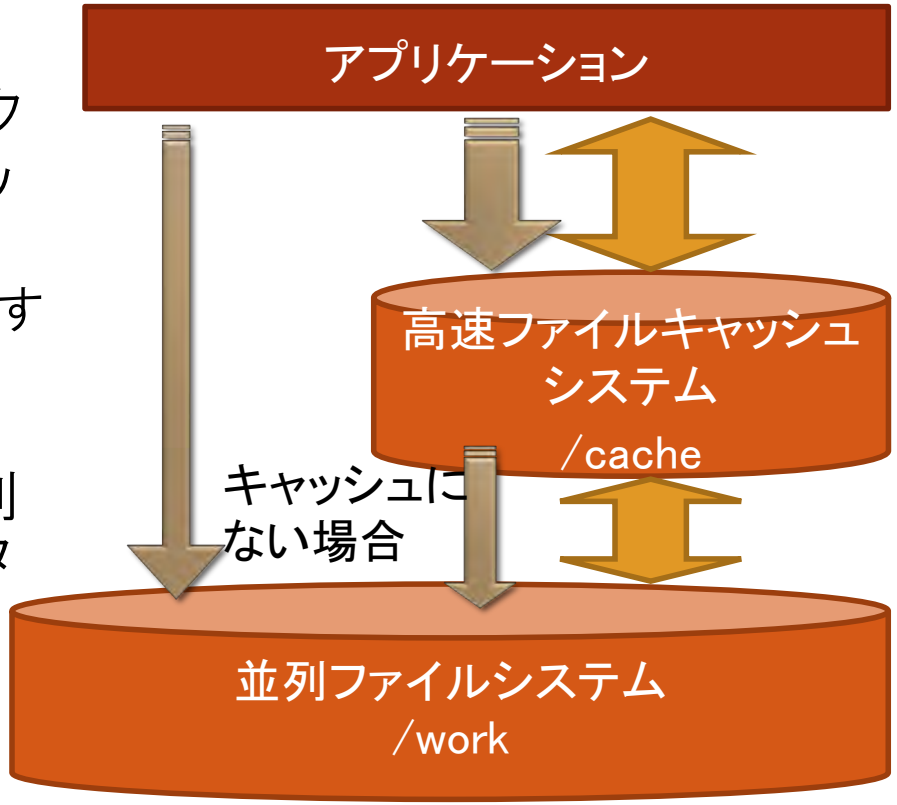
Collective IOの利点

- MPIプロセス数が増えると細かなIOが増加
  =>Collective IOによってIOをまとめる効果



高速ファイルキャッシュの（簡単な）使い方

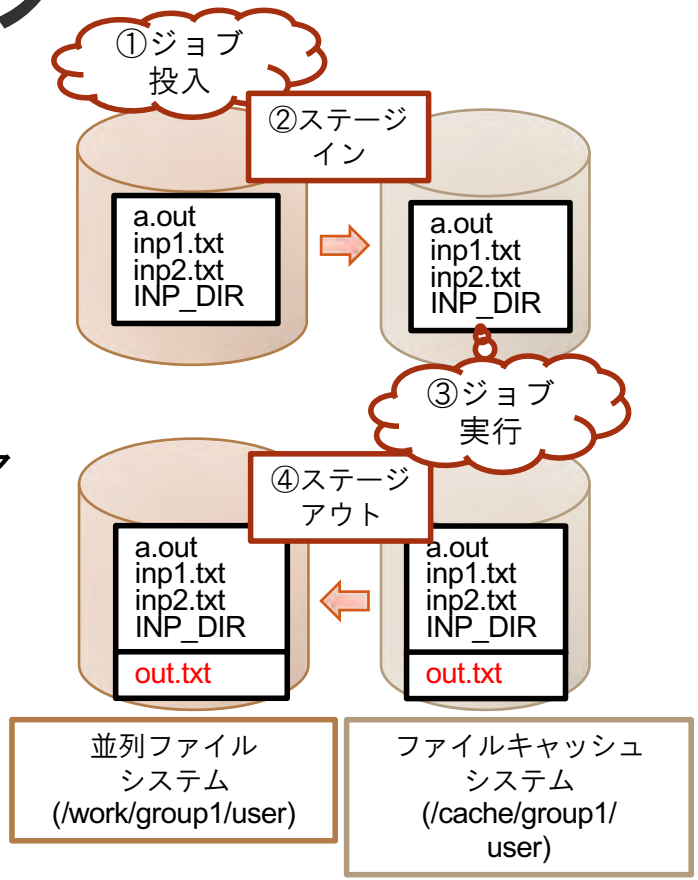
- /work の代わりに /cache にアクセスすると高速ファイルキャッシュ経由
  - 以下の2つは同一のファイルを指す
    /work/group1/user/file1
    /cache/group1/user/file1
- 高速ファイルキャッシュと並列ファイルシステムの間のデータ移動
  - ステージング



ファイルステージング

ジョブスケジューラと連携、高速ファイルキャッシュにステージング

- ステージインリスト、ステージアウトリストをファイルに記述: input_file.txt, output_file.txt
  - ジョブ投入時：並列ファイルシステム
    ⇨ファイルキャッシュ システム
  - ジョブ終了時: ファイルキャッシュシステム　並列ファイルシステム



ジョブスクリプトの記述例

```
#!/bin/bash
#PJM –L rscgrp=regular-cache
#PJM –L node=16
#PJM –g group1
#PJM –x STGIN_LIST=input_files.txt
#PJM –x STGOUT_LIST=output_files.txt

cd $PJM_O_CACHEDIR
prestage-wait
mpiexec.hydra –n ${PJM_MPI_PROC} ./a.out
```

対応する/cacheディレクトリに移動

ステージイン完了を待つ
300秒でタイムアウト

ステージングファイルの指定

- テキストファイルにファイル名を列挙

```
$ cat input_file.txt
```

```
# comment
/work/group1/userA/filename1   # 絶対パス
Filename2                        # 相対パス
file?.*                          # ワイルドカード (*, ?)
output_dir/                      # ディレクトリ指定、含まれるファイル全て
```

ステージアウト後の確認（重要！）

- ステージアウト処理は非同期に実行されます

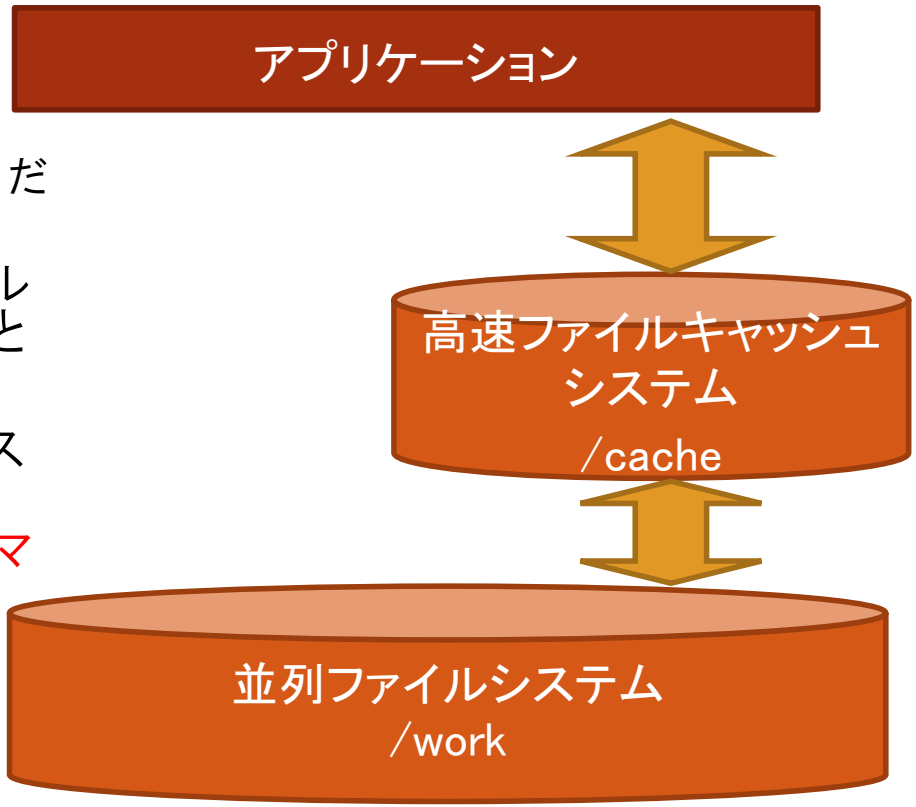
⇨ジョブ終了後でもステージアウトが終わっていない可能性があります

- ステージアウトの完了を待機（ブロック）
  $ sync-wait -f output_files.txt
- ステージアウトの状況を確認
  $ sync-list -f output_files.txt -v

```
Status          Clean          Dirty   Pending
Unsync              0    1,048,576         0
/work/group1/username/filename.1
Sync        1,048,576              0         0
/work/group1/username/filename.2
```

注意点

- ステージインしなくても /cache で参照はできます
  - 実際にキャッシュになければ /work の内容がアクセスされます（ただし遅い）
- ステージアウトしていないファイルは /work ではサイズ 0 のファイルとなります
- [重要] 高速ファイルキャッシュシステムを利用したファイルを削除するにはime-rmコマンドを用いる必要があります
  - ステージアウトしたファイルも対象です
  - ジョブスクリプト中 /cache 上のファイルは直接 rm しても問題ありません



主なユースケース

- チェックポインティング、各反復での配列データ出力
  - 特に、単一ファイル（single shared file）
  - ノード数を増やすと性能がスケール（出力時間が一定）
- 大規模データ解析
  - 大規模データを事前にステージング
- 大規模一時ファイル
  - ステージアウトせず不要時に直接削除
  - CACHE_TMPDIR環境変数を利用
    - 実際には /cache/tmp/ジョブ番号/ ディレクトリが用意される
  - 例： export TMPDIR=$CACHE_TMPDIR

効率的な利用のために

- ブロックサイズは大きく（1MB以上）
  - Fortranでは関係ありませんが、Cではwrite, readの単位を1MB以上としてください
- ファイルサイズを大きく（1GB以上）
  - 複数ファイルに対する同時I/Oの総量でもよい
  - 16TBを超えるファイルの場合は、あらかじめ並列ファイルシステムのストライプ設定が必要です
    - Lustre上の物理的な1ファイルは16TBまで

⇨ ストライプ設定し16TB未満に分割しないとステージアウトできなくなります

演習

- 実際のファイルIOを測定してみよう

- MPIIO
  - File Per Process (FPP)
  - Single Shared File (SSF)
    - block書き出し
    - stripe or cyclic書き出し
  - IME

コミュニケータ
グループ
トポロジ

コミュニケータ

- MPIにおいて，操作の対象になるMPIプロセスの集合を表すもの
- 定義済みのコミュニケータ
  - **MPI_COMM_WORLD** : MPI実行中の全プロセス
  - **MPI_COMM_SELF**: 自分自身のプロセス
- コミュニケータの種類
  - **イントラコミュニケータ (Intra-communicator)**
    - お互いに通信が可能，集団通信操作が可能
    - 通常使うのはこちら
  - インターコミュニケータ (Inter-communicator)
    - 異なるイントラコミュニケータに属するプロセス間での通信に用いる

コミュニケータの作成・開放

1. 既存のコミュニケータを利用して作る
   - MPI_Comm_dup, MPI_Comm_idup : 既存のコミュニケータを複製
   - MPI_Comm_split : 既存のコミュニケータを元に分割

2. グループを使って新たにコミュニケータを作る
   - MPI_Comm_create

- コミュニケータの削除
  - MPI_Comm_free

コミュニケータの分割

- プロセッサ群を分割したい場合、MPI_Comm_split 関数を利用
- ierr = MPI_Comm_split (comm, color, key, newcomm);
  - MPI_Comm comm (IN) : 元になるコミュニケータ
  - int color (IN): 同じcolorを持つランクが同じコミュニケータに入る
  - int key (IN): keyの小さいプロセスから，0から順にランクを割り当てる
  - MPI_Comm* newcomm (OUT): 新しいコミュニケータ
- commから，同じcolorを持つ複数のコミュニケータに分割
- 各newcommにはプロセスの重複はない
- **comm中の全プロセスが必ず実行すること**

MPI_Comm_splitの例

ランク	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
color	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
key	7	8	3	2	1	5	4	2	9	1	2	3	0	0	4	2

MPI_Comm_split

new-comm	(comm0)				comm1				comm2				comm3			
new-commにおけるランク	2	3	1	0	0	3	2	1	3	0	1	2	0	1	3	2
													1	0	3	2

グループからコミュニケータを生成(1)

- グループは，まずコミュニケータから生成する必要がある

- ierr = MPI_Comm_group(comm, group);

  - MPI_Comm comm (IN): 元になるコミュニケータ
  - MPI_Group* group (OUT): 得られたグループ

グループからコミュニケータを生成(2)

- 新しいグループを生成，ここでは部分集合を作成
- ierr = MPI_Group_incl(group, n, ranks, newgroup);
  - MPI_Group group (IN): 元になるグループ
  - int n (IN): 新しいグループに含まれるメンバ数
  - int ranks[] (IN): グループに含まれるランクのリスト（配列）
  - MPI_Group* newgroup (OUT): 得られた新しいグループ
- 他にもグループを編集する手段はいろいろある
  - MPI_Group_union, MPI_Group_intersection, MPI_Group_difference
  - MPI_Group_excl, MPI_Group_range_{incl,excl},…

グループからコミュニケータを生成(3)

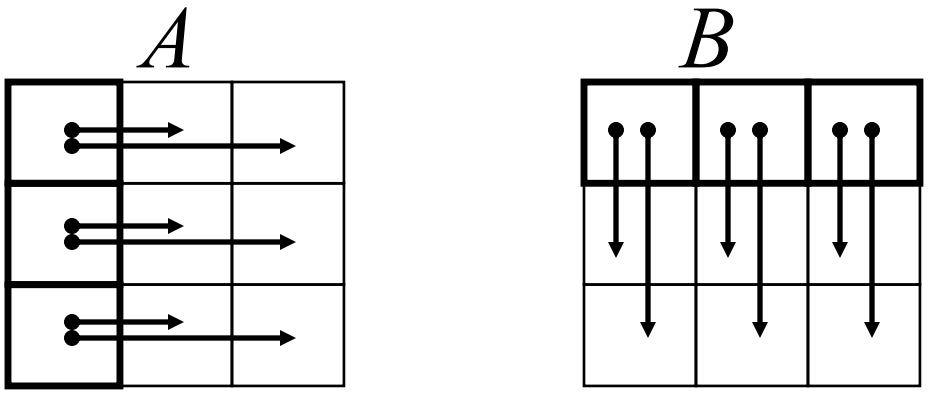
- グループからコミュニケータを生成
- ierr = MPI_Comm_create(comm, group, newcomm);
  - MPI_Comm comm (IN): 元になるコミュニケータ
  - MPI_Group group (IN): 部分集合グループ，commに含まれている必要がある
  - MPI_Comm* newcomm (OUT): 得られたコミュニケータ
- **comm中の全プロセスが必ず実行すること**

実例：SUMMAによる行列積

- SUMMA（Scalable Universal Matrix Multiplication Algorithm）
  - R.Van de Geijnほか、1997年
  - 同時放送（マルチキャスト）のみで実現

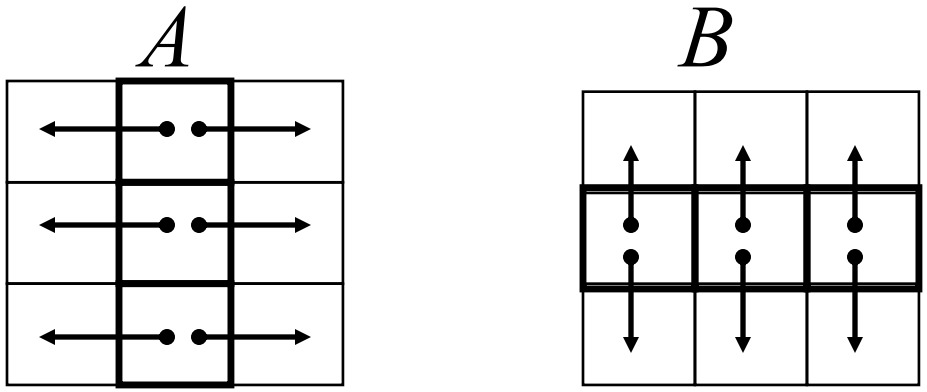
SUMMAアルゴリズムの概略

- 第一ステップ



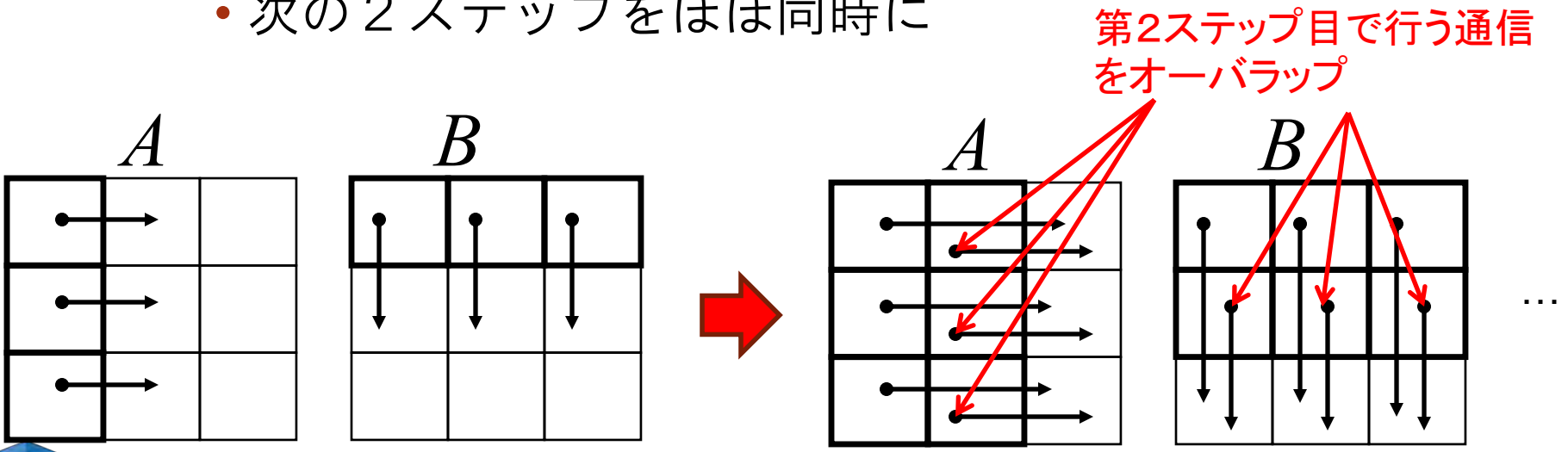
これを n/p

- 第二ステップ



SUMMA

- 特徴
  - 同時放送をブロッキング関数（例．MPI_Bcast）で実装すると、同期回数が多くなり性能低下の要因になる
  - SUMMAにおけるマルチキャストは、非同期通信の１対１通信（例．MPI_Isend）で実装することで、通信と計算のオーバラップ（***通信隠蔽***）可能
    - 次の２ステップをほぼ同時に



トポロジ

- MPIプロセス同士の接続情報
- 二次元以上の直交座標系に対してプロセスマッピングする際に使用すると便利
  - 各次元へのプロセス割り当て数を自動決定
  - 送受信先ランクの自動決定

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

プロセス数の割り当て

- 次元数を与えると適切なプロセス数を決定し、コミュニケータを割り当て
- ierr = MPI_Dims_create(nnodes, ndims, dims);
  - int nnodes (IN): 全プロセス数
  - int ndims (IN): 座標系の次元
  - int dims[] (INOUT): 各次元ごとのプロセス数
- 各次元ごとのプロセス数に非零数があったら尊重する

次元情報を持つコミュニケータ作成

- ierr = MPI_Cart_create(comm_old, ndims, dims, periods,reorder, comm_cart);

  - MPI_Comm comm_old (IN): 元にするコミュニケータ
  - int ndims (IN): 座標系の次元
  - int dims[] (IN): 各次元の次数 (ndims次元分)
  - int periods[] (IN): 周期境界かどうか(ndims次元分)
  - int reorder (IN): ランクを並べ替えるかどうか
  - MPI_Comm *comm_cart (OUT):新しいコミュニケータ

- comm_oldに属する全プロセスが呼ぶ必要がある
- dimsにはMPI_Dims_create()で得られたものをそのまま渡せば良い

座標を得る

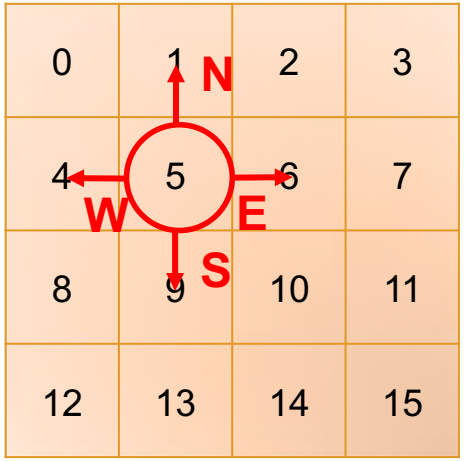
- ierr = MPI_Cart_coords(comm, rank, maxdims, coords);
  - MPI_Comm comm (IN): コミュニケータ
  - int rank (IN): ランク番号 (comm内の）
  - int maxdims (IN): 次元数
  - int coords[] (OUT): 座標(の配列)
- commはMPI_Cart_createにおいて作成したコミュニケータ
  - ランク番号は(create時にreorder=1なら)以前と変わる可能性があることに注意 => MPI_Comm_rankで改めてランク番号を取得

通信相手ランクを得る

- ierr = MPI_Cart_shift(comm, direction, displ, source, dest);

  - MPI_Comm comm (IN): コミュニケータ
  - int direction (IN): 座標の次元番号
  - int displ (IN): 相手までの距離
  - int * source (OUT): (自分にとっての)送信元ランク
  - int * dest (OUT): 送信先ランク

- 二次元の場合、以下のようにすれば隣接ノードのランクが得られる
  - MPI_Cart_shift(comm, 0, 1, &north, &south);
  - MPI_Cart_shift(comm, 1, 1, &west, &east);
  - 次元順の取り方はFortran的



Hybrid並列

MPIのマルチスレッド対応

- アプリケーション中では1スレッドしか許さない
  - MPI_THREAD_SINGLE
- マルチスレッド対応
  - MPI_THREAD_FUNNELED
    - マスタースレッドのみがMPIを呼ぶ
  - MPI_THREAD_SERIALIZED
    - 誰でもMPIを呼べるが，内部では逐次化される
  - MPI_THREAD_MULTIPLE
    - 完全なマルチスレッド動作

各モードにおける記述

- SingleではParallelリージョンから出なければいけない
- Funneledではmasterだけが呼べる
- Serialized, Mutipleは誰が呼んでもいい
  - タグで通信を区別する必要

Single

```
#pragma omp parallel
{
....
}
   MPI_Send( ... );
#pragma omp parallel
 {
....
}
```

Funneled

```
#pragma omp parallel
{
  ....
#pragma omp master
  MPI_Send( ... );
  ....
}
```

Master onlyと似ているが
parallel節を閉じなくていい

Serialized, Multiple

```
#pragma omp parallel
{
  ....
  MPI_Send( ... );
  ....
}
```

マルチスレッドMPIの初期化

- MPI_Init()の代わりに、MPI_Init_thread()を使用

- C言語:

```
int provided;
MPI_Init_thread(&argc,&argv,MPI_THREAD_FUNNELED,
&provided);
```

- Fortran

```
integer required, provided
required=MPI_THREAD_FUNNELED
call MPI_Init_thread(required, provided, ierr)
```

Probe / Iprobe

- 実際に受信する前に受信データをチェックしてステータスのみを取得
- ierr = MPI_Probe (source, tag, comm, status)
- ierr = MPI_Iprobe (source, tag, comm, flag, status)
  - int source (IN), int tag (IN), MPI_Comm comm (IN), MPI_Status *status (OUT): MPI_Recvと同様
  - int *flag (OUT): flag=TRUEのときstatusを利用可能
- メッセージサイズを知って受信バッファサイズを確保
- nonblocking通信の加速
  - 実装によっては，MPI_Waitするまで通信が起こらない
  - MPI_Probeによってバックグラウンドの処理が進む

MPI_Probeの利用例

```
MPI_Status status;
int count;
int *rbuf;

MPI_Probe( MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status );
MPI_Get_count( &status, MPI_INT, &count );
// statusから受信データ数を取得
MPI_Alloc_mem( count*sizeof(int), MPI_INFO_NULL, rbuf );
// メモリ確保
MPI_Recv( rbuf, count, MPI_INT, status.MPI_SOURCE, status.MPI_TAG,
MPI_COMM_WORLD, MPI_STATUS_IGNORE );
//  実際の受信, Fortranでは代わりに status(MPI_SOURCE), status(MPI_TAG)
```

マルチスレッドにおけるprobe

- MPI_Probe, MPI_Iprobeでは，スレッド間で取りちがえる可能性
- MPI_Mprobe, MPI_Improbeおよび MPI_Mrecv, MPI_Imrecvを使う

- ierr = MPI_Mprobe(source, tag, comm, message, status)
- ierr = MPI_Improbe(source, tag, comm, flag, message, status)
- ierr = MPI_Mrecv (buf, count, datatype, message, status)
- ierr = MPI_Imrecv(buf, count, datatype, message, request)

  - MPI_Message *message : Mrecv/Imrecvのためのメッセージハンドル

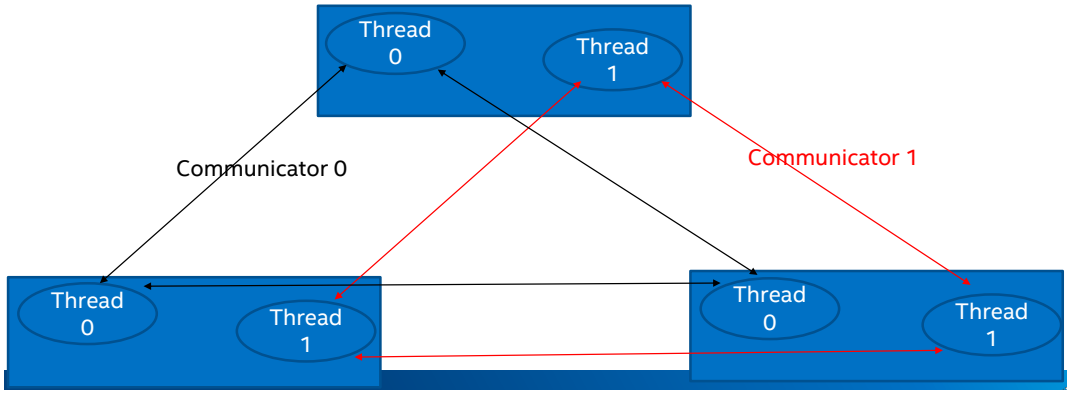
MPI_THREAD_MULTIPLEでの注意

- デフォルトでサポートしていない実装系もある
  - ビルド時に明示的に有効にする：Open MPI
  - MPI_Init_thread の providedで確認
- 性能を出すのが難しい面がある
- 集団通信には使えない
- 各スレッドで明示的にタグやコミュニケータを使い分けることで通信を識別する必要がある

Multiple Endpoint拡張

- Intelによる拡張、条件を制限することで高速化
  - 2018.3以降で利用可能(?)
- export I_MPI_THREAD_SPLIT=1
- export I_MPI_THREAD_RUNTIME=openmp
- export PSM2_MULTI_EP=1
- export I_MPI_THREAD_MAX=<通信に使う最大スレッド数>
  - プログラム用の指定は別途必要

Rule: Thread N in rank A has to communicate with Thread N in rank B in the same communicator.



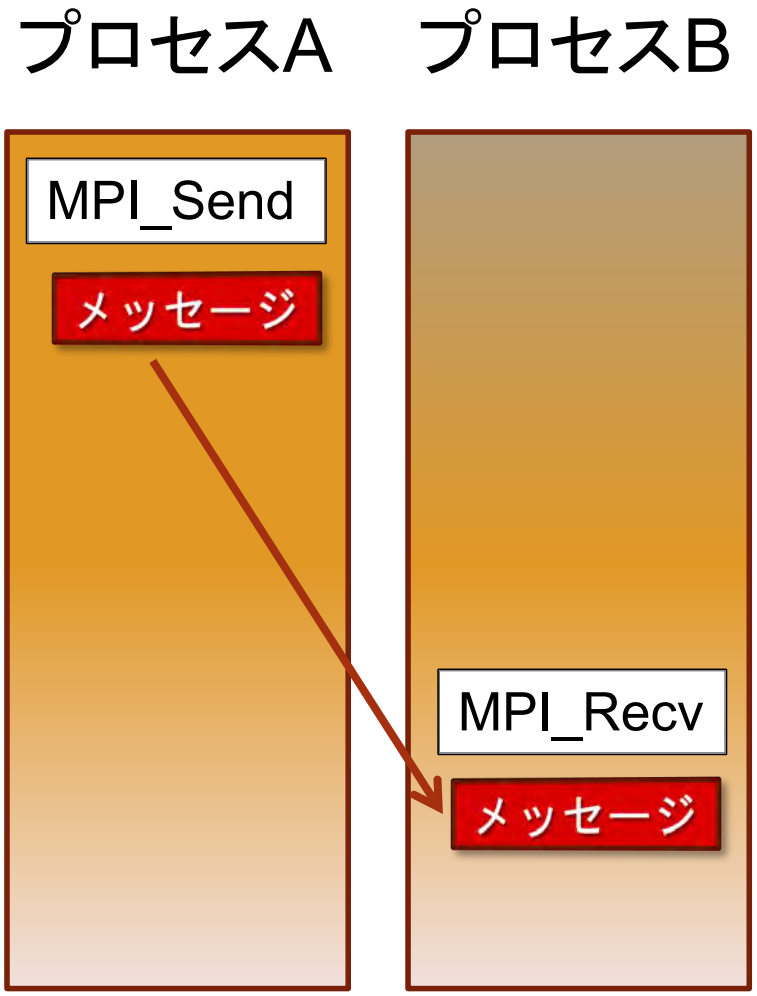
片側通信 / One sided communication

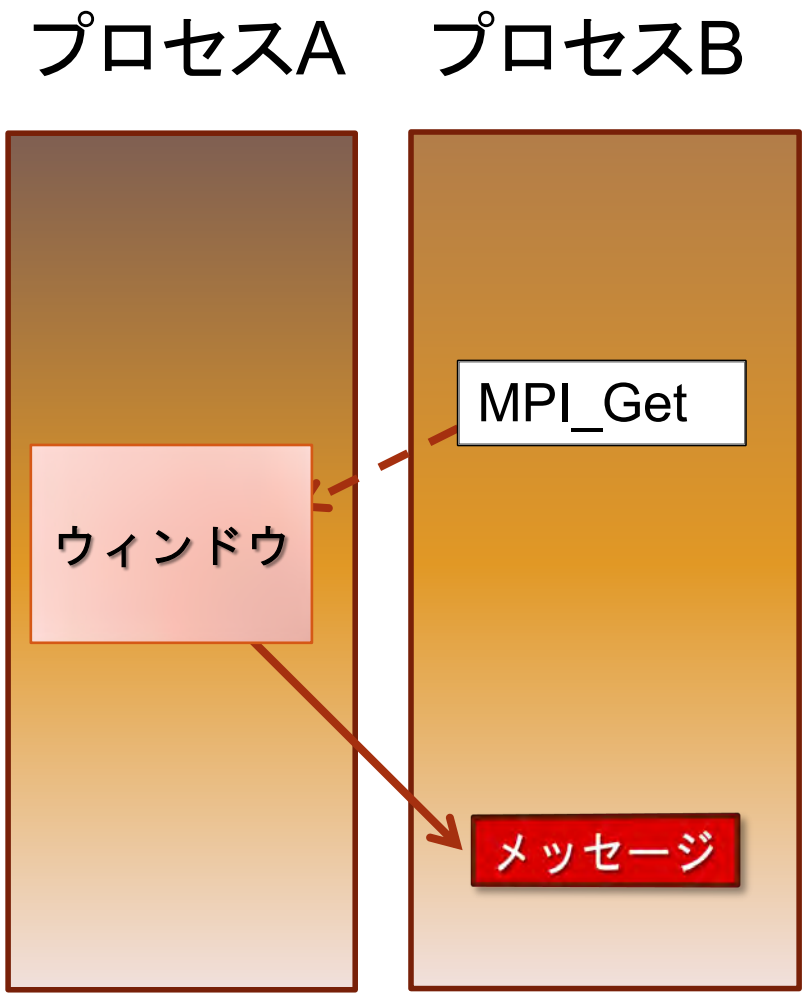
片側通信

- 一方のプロセスのみで通信が完結
  - 通信相手の状態には無関係に記述できる

- 片側通信の利点
  - ハードウェア通信機構 Remote DMA (RDMA)との親和性
  - データコピーの削減
  - 同期コストの削減

１対１通信 vs 片側通信





基本的な使い方の流れ

1. Windowオブジェクトの生成，割り当て
2. エポックの開始
3. ターゲットへのアクセス
4. 同期
5. エポックの終了
6. Windowオブジェクトの解放

Windowオブジェクトの生成

- リモートメモリアクセス可能なWindowを定義
- すでに配列がある場合
- ierr = MPI_Win_create(base, size, disp_unit, info, comm, win);
  - void *base (IN): ウィンドウにしたいメモリの先頭アドレス
  - MPI_Aint size (IN): サイズ（バイト数）
  - int disp_unit (IN): Put, Get等で指定されるデータのサイズ
  - MPI_Info info (IN): ヒント情報（後述）
  - MPI_Comm comm (IN): コミュニケータ
  - MPI_Win *win (OUT): 得られるWindowオブジェクト
- コミュニケータの全メンバーが呼ぶこと

Windowオブジェクトの割り当て

- リモートメモリアクセス可能なWindowを定義
- 新たに配列を確保する場合
- ierr = MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win);
  - MPI_Aint size (IN): サイズ（バイト数）
  - int disp_unit (IN): Put, Get等で指定されるデータのサイズ
  - MPI_Info info (IN): ヒント情報（後述）
  - MPI_Comm comm (IN): コミュニケータ
  - void *baseptr (OUT): ウィンドウになるメモリの先頭アドレス
    Fortranの場合、type(C_PTR)でないといけない
    （演習の解答例を参照）
  - MPI_Win *win (OUT): 得られるWindowオブジェクト
- コミュニケータの全メンバーが呼ぶこと

Windowオブジェクトの解放

- ierr = MPI_Win_free(win);

  - MPI_Win *win (INOUT): 得られるWindowオブジェクト

MPI_Put: リモートメモリ書き込み

- ierr = MPI_Put(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win);

  - void *origin_addr (IN): 自プロセスの先頭アドレス 書き込みたいデータ
  - int origin_count (IN): 自プロセスのデータ個数
  - MPI_Datatype origin_datatype (IN): 自プロセスのデータ型
  - int target_rank (IN): ターゲットのランク
  - MPI_Aint target_disp (IN): ターゲットのウィンドウからの位置
  - int target_count (IN): ターゲットに書き込むデータ個数
  - MPI_Datatype target_datatype (IN): ターゲットのデータ型
  - MPI_Win win (IN): ウィンドウオブジェクト

  - target_addr = window_base+target_disp×disp_unit (Window生成時)

MPI_Get: リモートメモリ読み出し

- ierr = MPI_Get(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win);
  - void *origin_addr (OUT): 自プロセスの先頭アドレス，読み出したデータが入る
  - int origin_count (IN): 自プロセスのデータ個数
  - MPI_Datatype origin_datatype (IN): 自プロセスのデータ型
  - int target_rank (IN): ターゲットのランク
  - MPI_Aint target_disp (IN): ターゲットのウィンドウからの位置
  - int target_count (IN): ターゲットに書き込むデータ個数
  - MPI_Datatype target_datatype (IN): ターゲットのデータ型
  - MPI_Win win (IN): ウィンドウオブジェクト

  - target_addr = window_base+target_disp×disp_unit (Window生成時)

その他の関数

- MPI_Accumurate
- MPI_Get_accumurate
  - リダクション演算に似ている
- MPI_Compare_and_swap
- MPI_Fetch_and_op
  - Atomic演算

同期

- エポック：メモリアクセスを許可する期間
- 同期関連の操作と関連

- アクティブターゲット：両方のプロセスが関与
  1. MPI_Win_fence
  2. MPI_Win_post, MPI_Win_start, MPI_Win_complete, MPI_Win_wait (PSCW)
- パッシブターゲット：オリジン側のプロセスだけ，ターゲット側は何もしない
  - MPI_Win_lock/MPI_Win_unlock, MPI_Win_flush

Fence

- バリア同期に類似，winに属するプロセスが全て呼び出し
- 様々な組み合わせ，高い頻度で通信がある場合
- ierr = MPI_Win_fence(assert, win);
  - int assert (IN): 通常は0
  - MPI_Win win (IN): ウィンドウオブジェクト

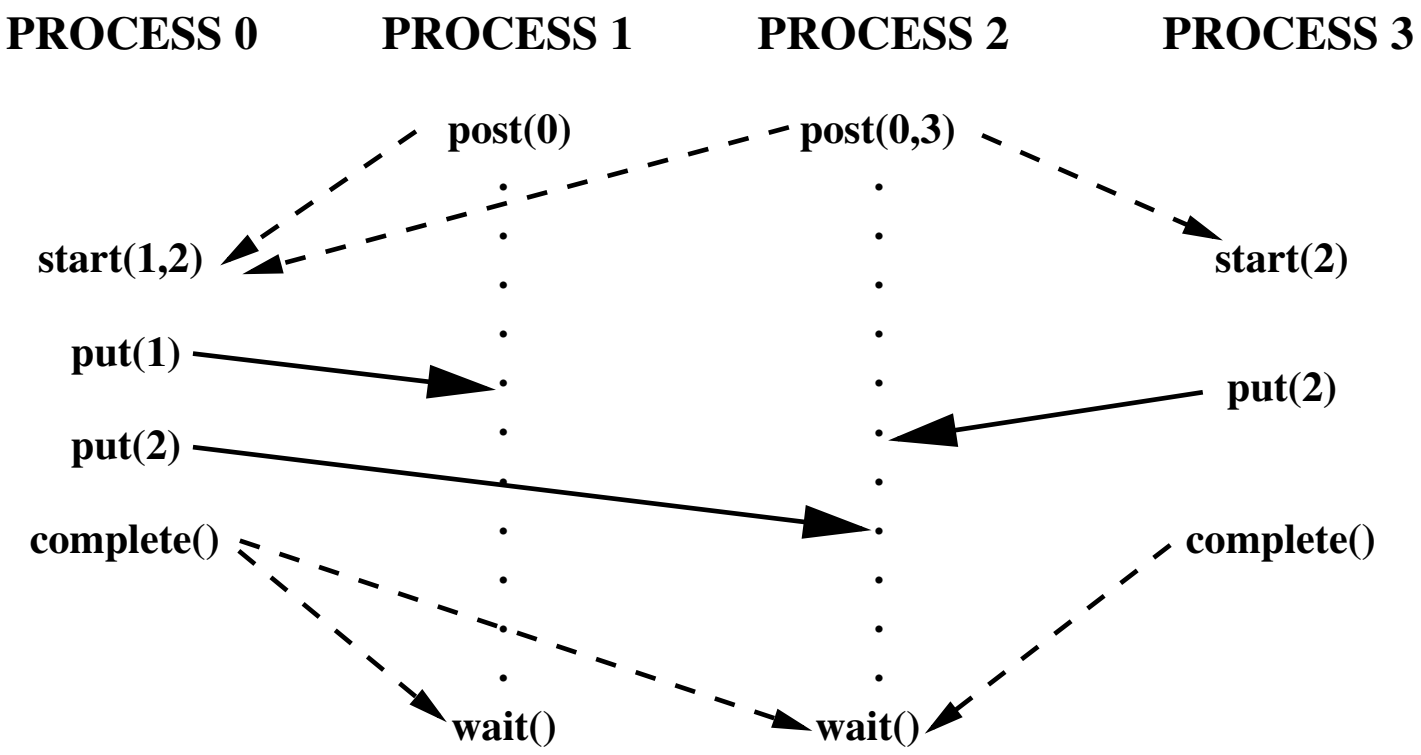
```
MPI_Win_fence(0, win);
if(rank == 0) MPI_Put( … , win);

MPI_Win_fence(0,win);
```

この間アクセス可能（エポック）

同期

PSCW同期

- 細かな同期の制御が可能
  - post-wait: ターゲット，start-complete: オリジン



Post, Start, Complete, Wait

- ierr = MPI_Win_post(group, assert, win);
- ierr = MPI_Win_start(group, assert, win);
- ierr = MPI_Win_complete(win);
- ierr = MPI_Win_wait(win);

  - MPI_Group group (IN): 公開する相手のグループ
  - int assert (IN): 通常は0
  - MPI_Win win (IN): ウィンドウオブジェクト

- win内のgroupメンバーで必要なものだけが呼べば良い

Lock/Unlock

- パッシブターゲットのエポックを定義

- ierr = MPI_Win_lock(lock_type, rank, assert, win);
  - int lock_type (IN): MPI_LOCK_EXCLUSIVE or MPI_LOCK_SHARED
  - int rank (IN): ターゲットのランク番号
  - int assert (IN): 通常は0
  - MPI_Win win (IN): ウィンドウオブジェクト

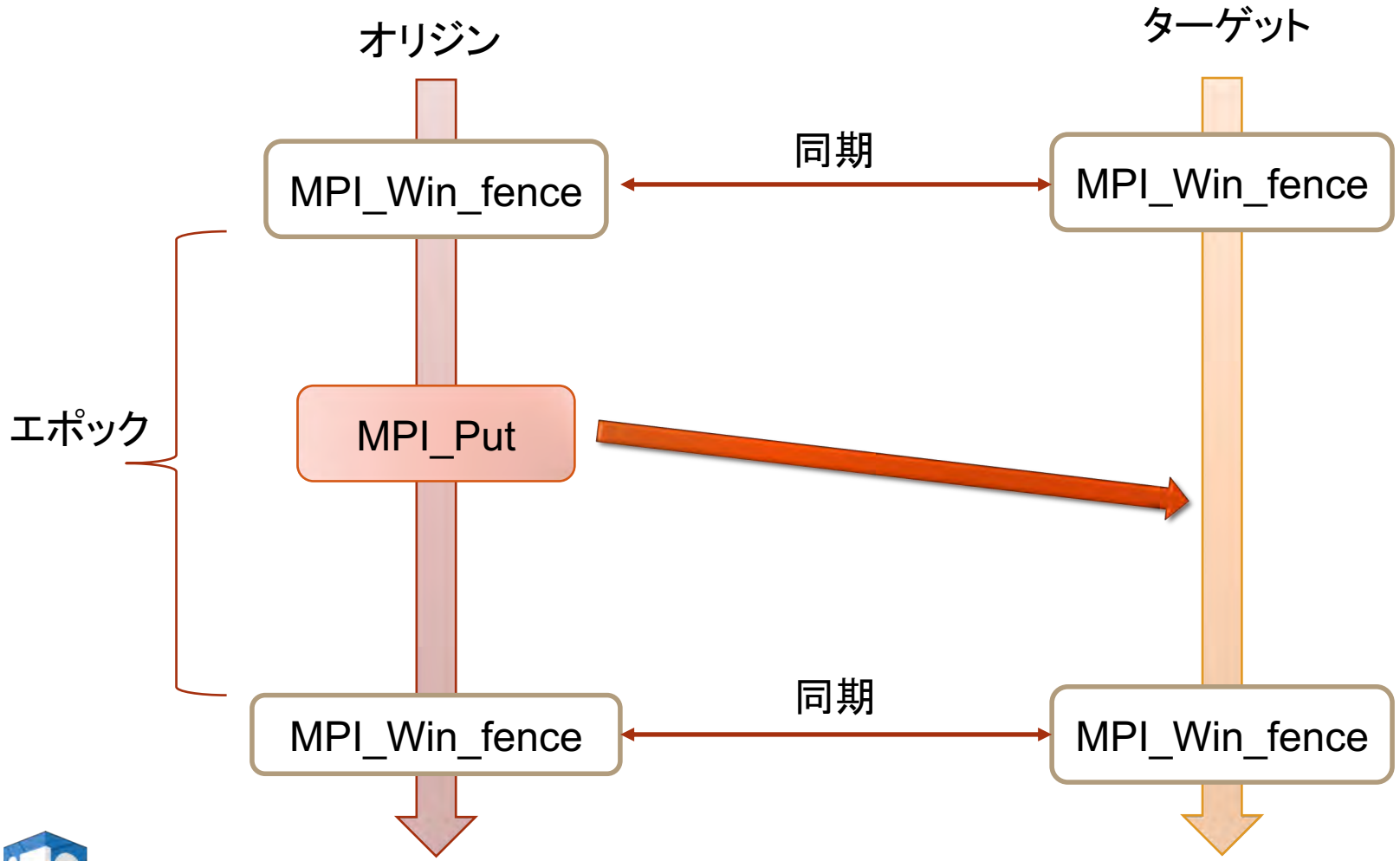
- ierr = MPI_Win_unlock(rank, win);

Flush

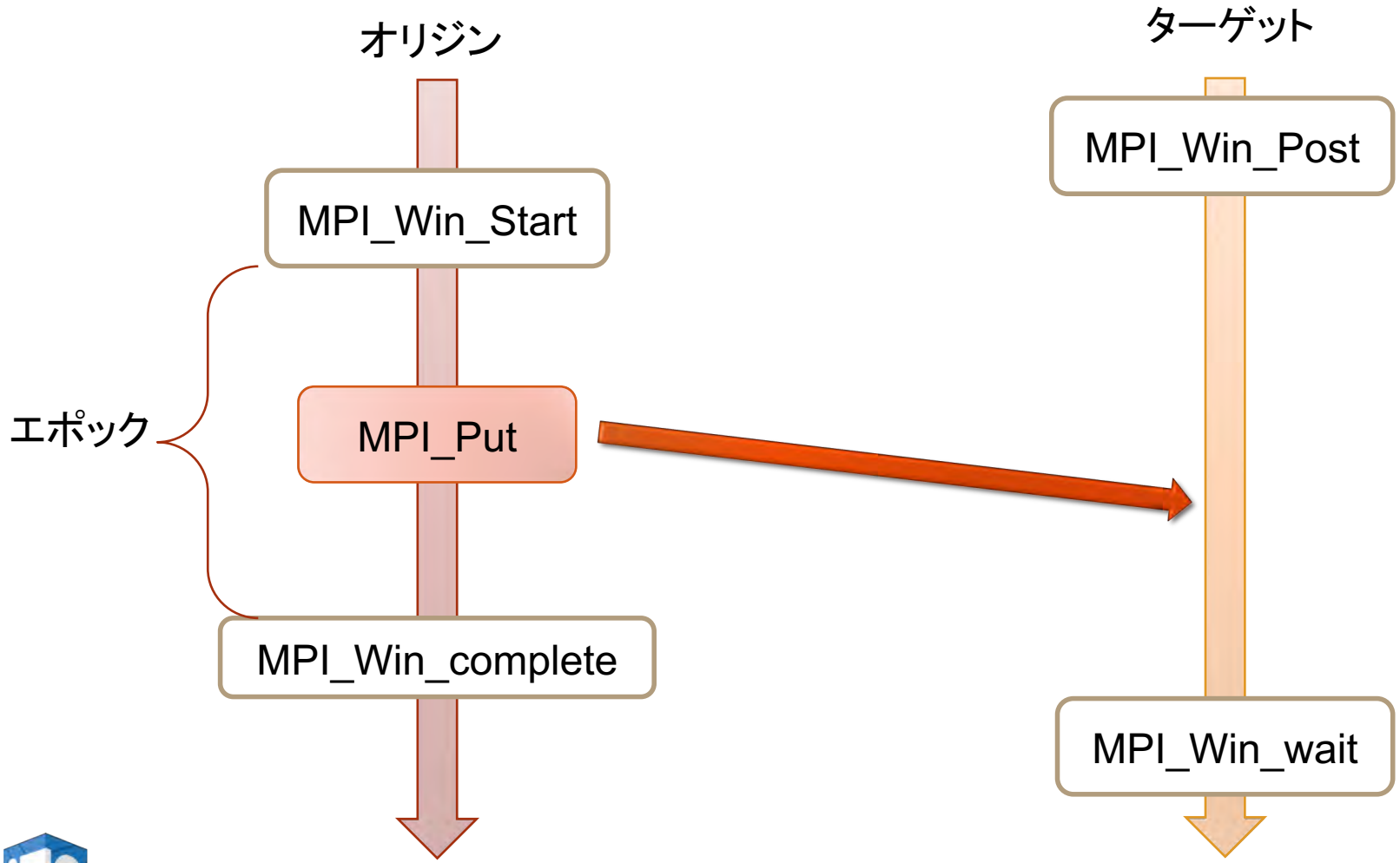
- パッシブターゲットのエポック内で利用可能
- 実行中の処理を完了させる
- ierr = MPI_Win_flush(rank, win);

  - int rank (IN): ターゲットのランク番号
  - MPI_Win win (IN): ウィンドウオブジェクト

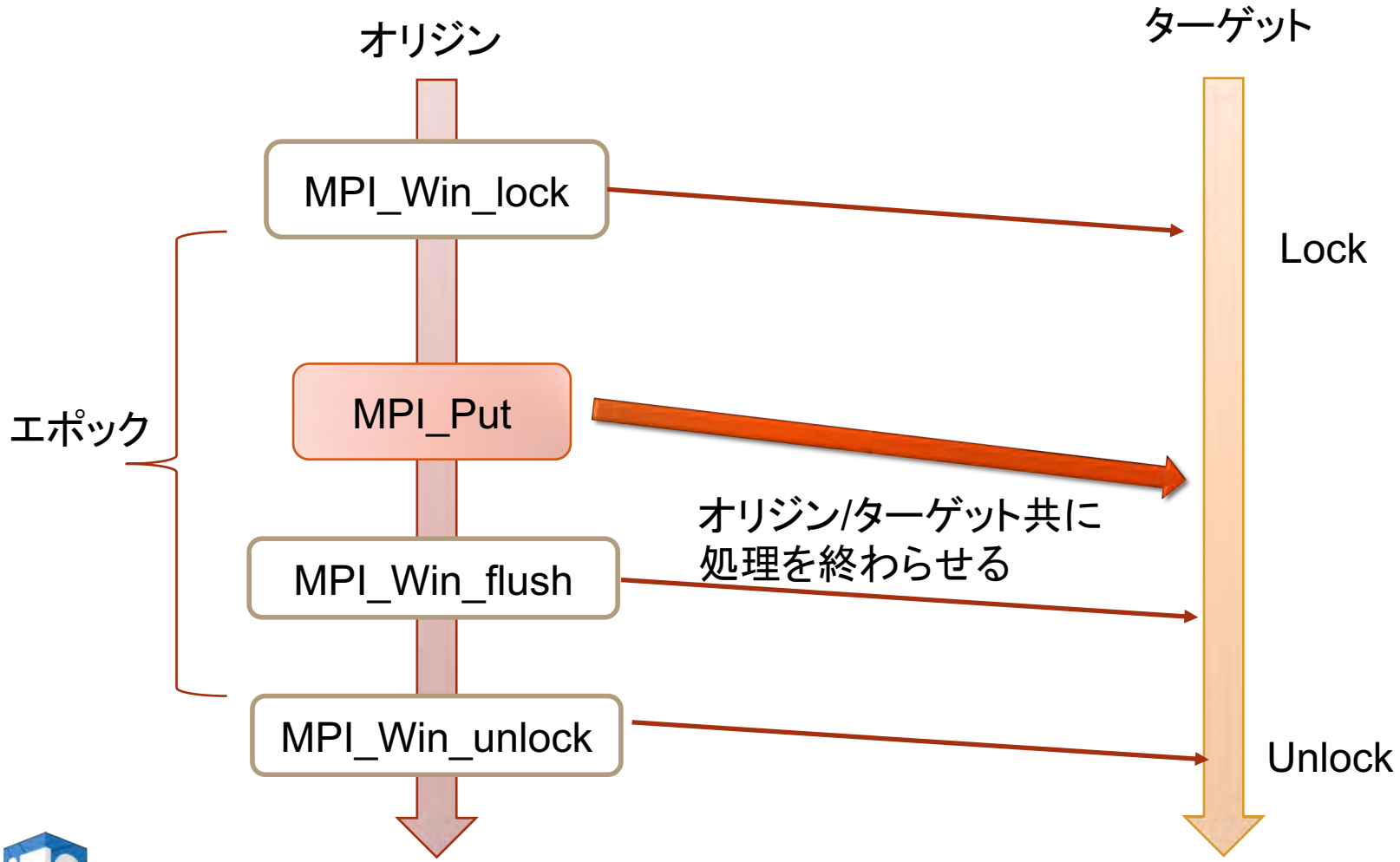
アクティブターゲット (Fence)



アクティブターゲット (PSCW)



パッシブターゲット (Lock)



演習

- 2つのプロセス間でデータ交換
  - Lock
  - Fence
  - PSCW

- 配列サイズ2N
- 後半のN個分を入れ替え

- MPI-advanced/onesided/