格子ゲージ理論のための汎用コード Bridge++ の開発

松古 栄夫 (Hideo Matsufuru)
高エネルギー加速器研究機構 計算科学センター

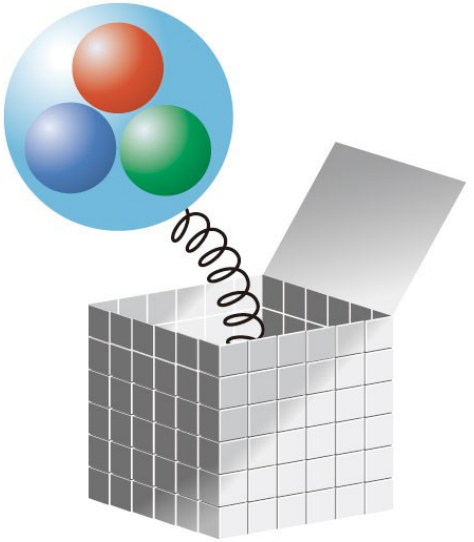
2019年度第2回計算科学フォーラム

2020年2月27日, 東京大学本郷キャンパス (東京会場)

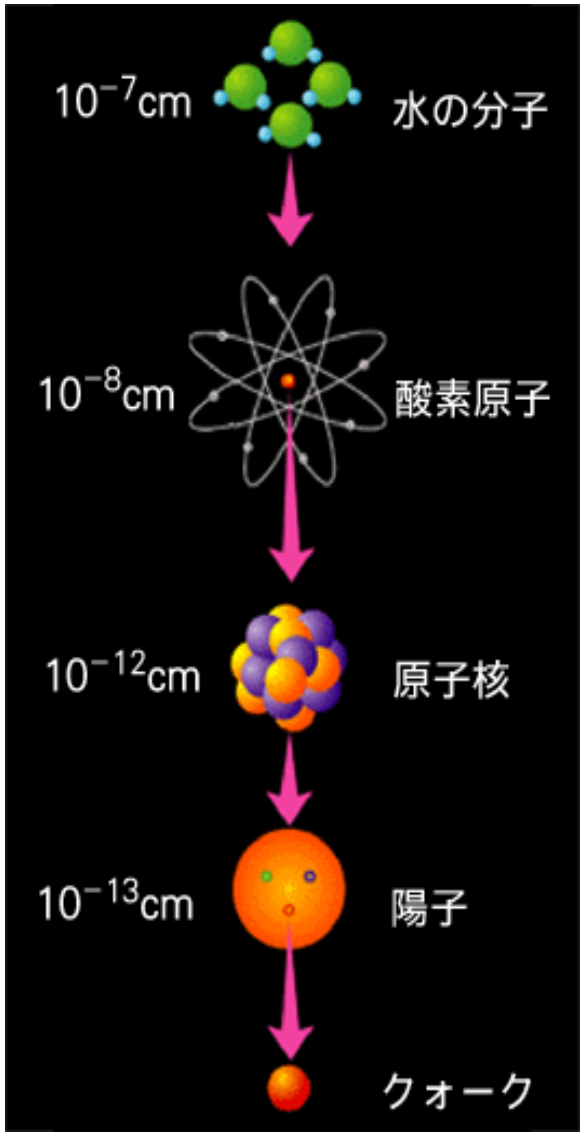
Contents

- Introduction
  - 素粒子物理の中でのQCD
  - 格子QCDシミュレーションの原理
- 汎用格子ゲージ理論コード Bridge++ の開発
  - 開発体制
  - オブジェクト指向へのアプローチ
  - Bridge++のコードデザイン
- 最新アーキテクチャへの対応
  - これまでのターゲットアーキテクチャ
  - Fugaku に向けて

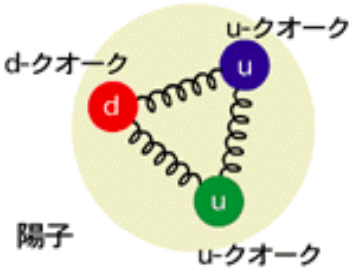
格子QCDの原理



クォークとハドロン



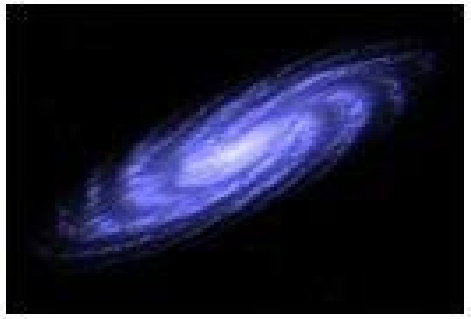
- クォーク
  - 陽子や中性子をつくる「素粒子」
  - ４つの相互作用すべてに関わる
  - 強い相互作用 ー 量子色力学(QCD)で記述
  - 電荷： 2/3 (u, c, t), -1/3 (d, s, b)
- ハドロン： クォークからなる粒子
  - クォークは単独では存在できない
  - 陽子： uud, 中性子： udd
  - 中間子はクォークと反クォークから
  - 加速器実験で多くのハドロンが発見
- 基礎理論(QCD)はわかっているが実際に解くのは困難



4つの相互作用

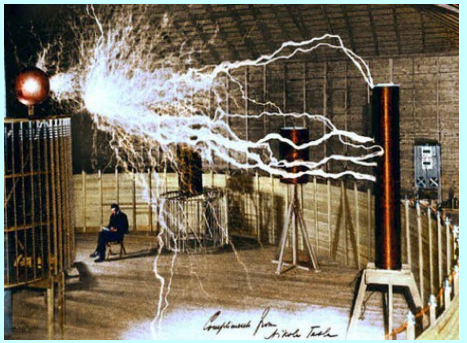
自然界には４つの力（相互作用）が知られている

重力



弱いけど引力しかない
遠くまで届く：天体や宇宙の構造を作る
最近の注目：重力波

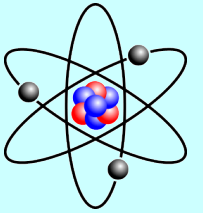
電磁気力

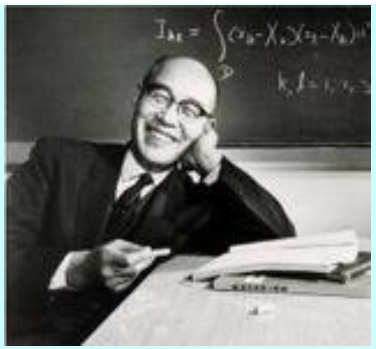


静電気や磁気、身の回りの現象
原子核と電子を結びつけ原子をつくる

統一された： 電弱理論

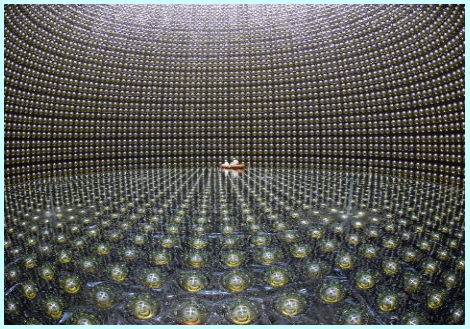
強い力





クォークを結び付けて陽子や中性子をつくる
陽子、中性子に働いて原子核をつくる

弱い力



陽子と中性子の変換（ベータ崩壊）
短い距離で働く
ニュートリノを放出

素粒子の標準模型

「標準理論」 現在の素粒子モデル – これで満足してる研究者は少ない

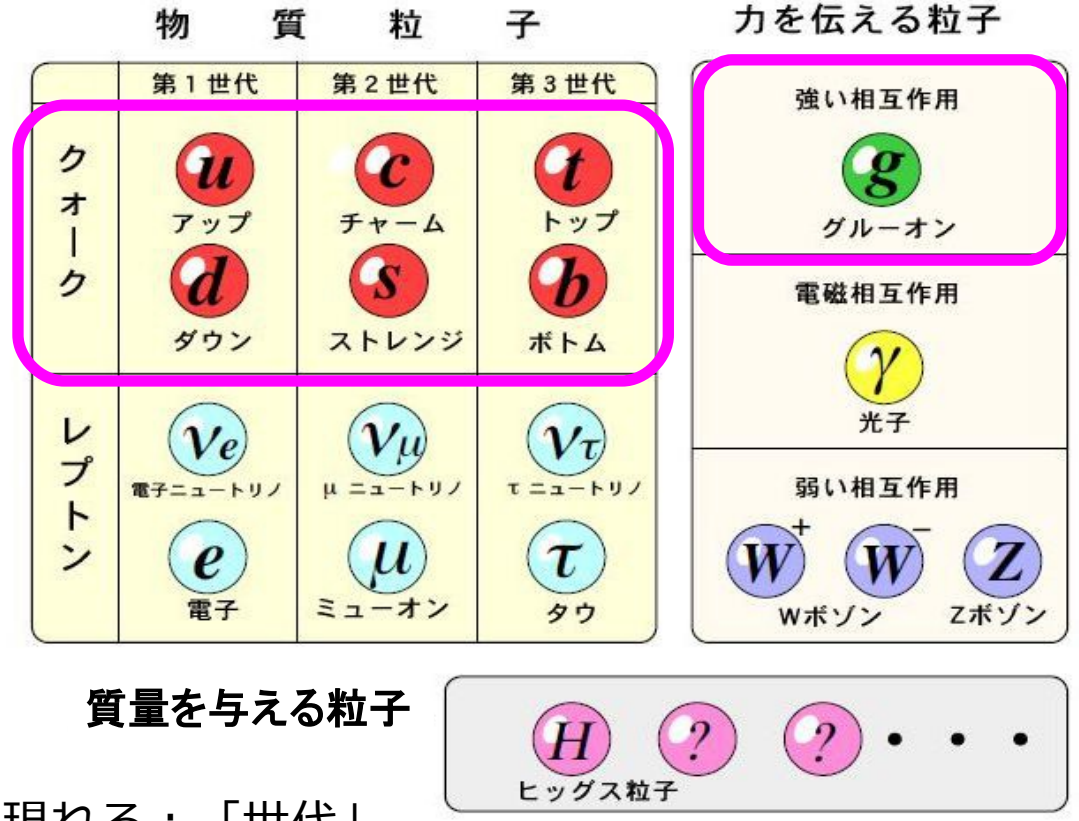
物質粒子はフェルミオン

- １つの状態には１粒子のみ
- クォーク （強い力あり）
- レプトン （強い力なし）

力を伝える粒子はボソン

- 一つの状態を粒子がいくつでも占めることができる

ヒッグス粒子：質量を与える



- 同じようなタイプの粒子が繰り返して現れる：「世代」
  - 質量は大きく違う
  - 3世代しかないらしい
- ニュートリノの性質にはまだ謎が多い – 質量を持つことが神岡実験で判明

量子色力学 (QCD)

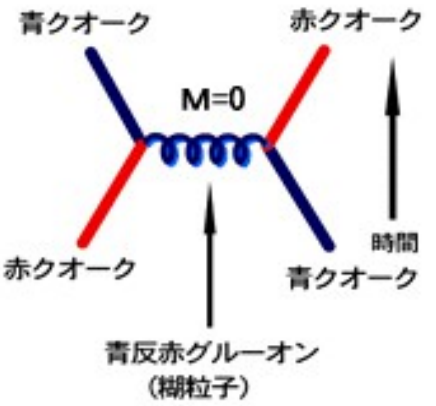
- 強い相互作用の基礎方程式 (Langrangian)はわかっている
  量子色力学 (Quantum Chromodynamics, QCD) という理論で記述

$$\mathcal{L}(x) = -\frac{1}{4}F^{a\mu\nu}(x)F^{a}_{\mu\nu}(x) + \bar{\psi}(x)(i\gamma_\mu D_\mu - m)\psi(x)$$

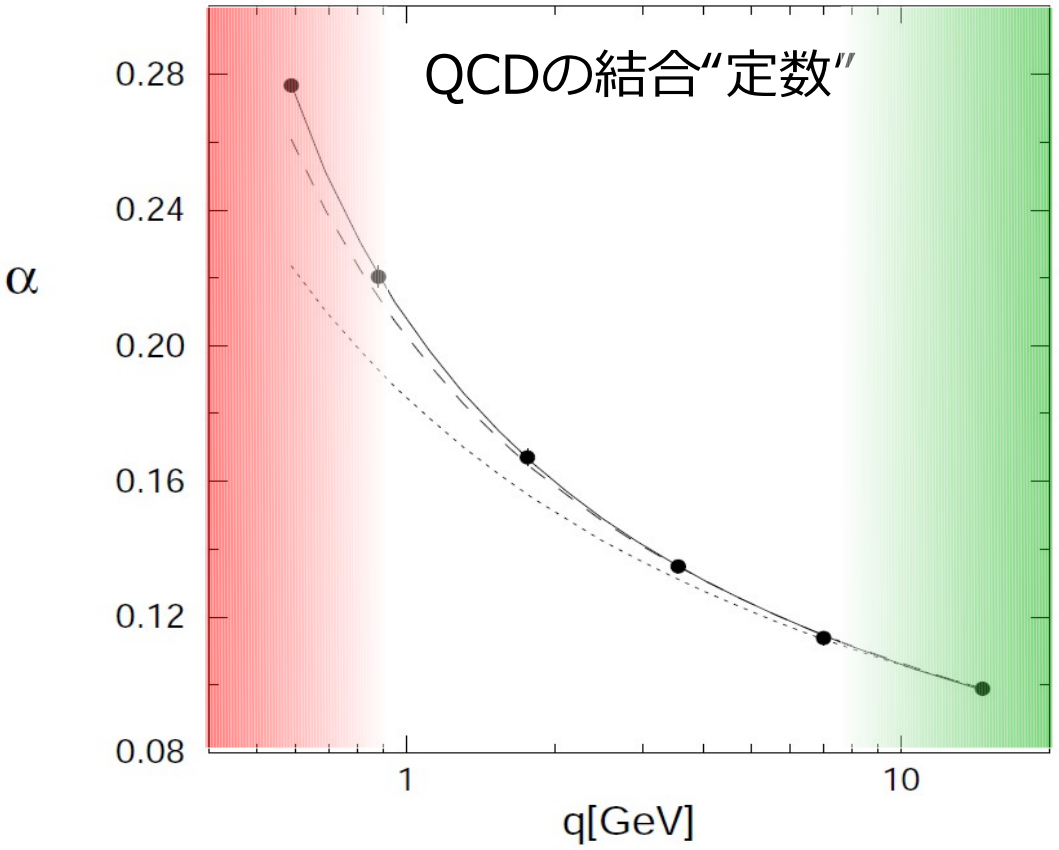
$$F^{a}_{\mu\nu} = \partial_\mu A^a_\nu - \partial_\nu A^a_\mu + g f^{abc} A^b_\mu A^c_\nu \qquad D_\mu = \partial_\mu + igA^a_\mu T^a$$

グルーオン場を記述　　クォークとグルーオンの相互作用

- 量子電磁気学 (QED) とよく似た形
  - QED は高精度で計算できる： 摂動論が有効
  - QCD は解析的に解くのは困難 (ミレニアム問題)
    - グルーオンの自己相互作用が原因
    - 摂動論は高エネルギー領域でしか使えない
  - 数値計算なら力ずくで計算できる（速い計算機が必要！）



QCDはなぜ難しい？



M.Luscher et al., Nucl. Phys. B 413, 481 (1994)

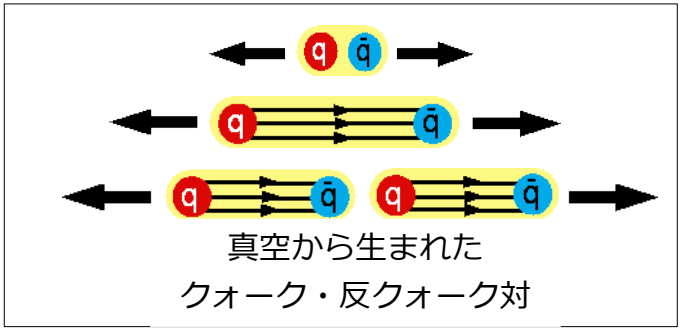
高エネルギー：「漸近的自由性」

- 高エネルギー(短距離)で結合が減少
- 摂動論が有効

2004年ノーベル賞

低エネルギーでは結合が増大

- 摂動論の展開が小さくなっていかず破綻
- 距離を離すほど、必要なエネルギーが増大



クォークを単独で取り出せない：「閉じ込め」

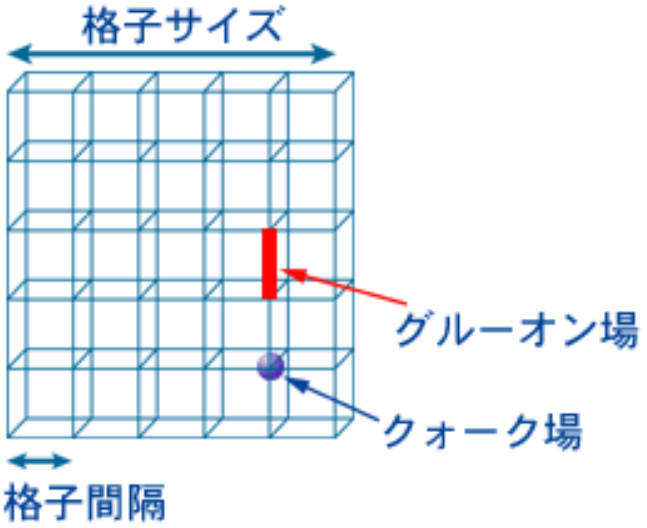
精密計算の必要性

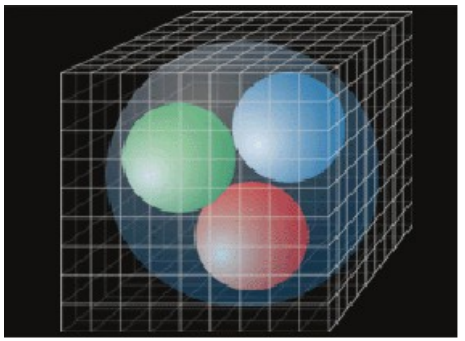
- 素粒子実験との比較
  - 標準理論の検証とそれを超える物理の探索
    - 標準理論の予言からのズレ → 未知の物理？
    - 標準理論を超える物理の候補の計算
    - テクニカラー、超対称性、etc.
- 有限温度・密度でのQCD相構造
  - 中性子星、超新星爆発など – 状態方程式が必要
  - 宇宙初期の相転移 – 重イオン衝突実験との比較
- 第一原理(QCD)からの核力の理解

精度のよいQCDからの予言が不可欠

格子上のQCD

- 時間と空間を4次元の格子で近似
  - Minkowski → Euclid 化
  - 有限の格子間隔と格子サイズ
    → 連続極限、熱力学極限を取る必要
- その上にクォークとグルーオンを乗せる
  - クォーク：サイト上の反交換 Grassmann 数
    → 数値的に扱えないので手で積分
  - グルーオン： リンク上の SU(3) 行列
- 経路積分で量子化
  - あらゆる「配位」について積分
  - Monte Carlo 法で「場の配位」を生成
  - 測定量を計算して統計処理 → 計算結果＋統計誤差





格子QCDシミュレーションの原理

- 物理量の期待値:

$$\langle O \rangle = \int \mathcal{D}U \mathcal{D}\phi^\dagger \mathcal{D}\phi \, O[U] \exp\left[-S_G[U] - \phi^\dagger D^{-1}[U]\phi\right]$$

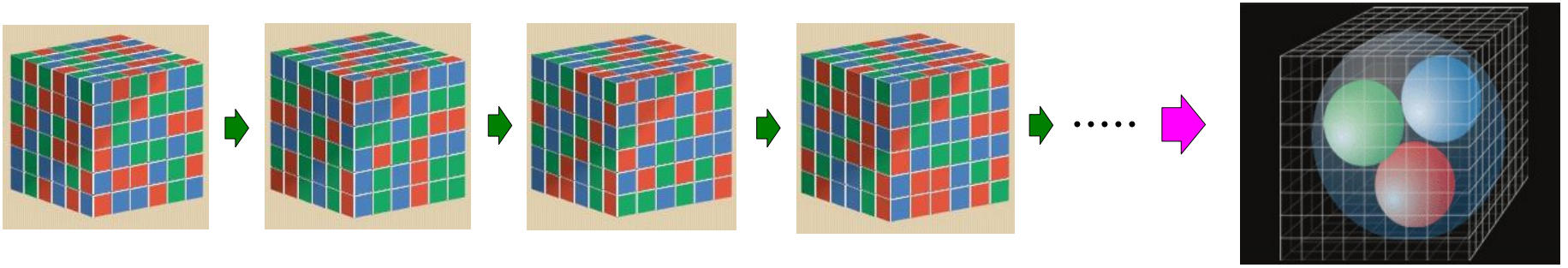
格子上の作用

グルーオン場U, 擬クォーク場φの配位についての汎関数積分

擬クォーク場の有効作用 (このDの逆を解くのに時間かかる)

- モンテカルロ法: グルーオン場の配位{U} (と擬クォーク場 Φ) を

$\exp\left[-S_G[U] - \phi^\dagger D^{-1}[U]\phi\right]$ の確率 (Boltzmann weight) で生成



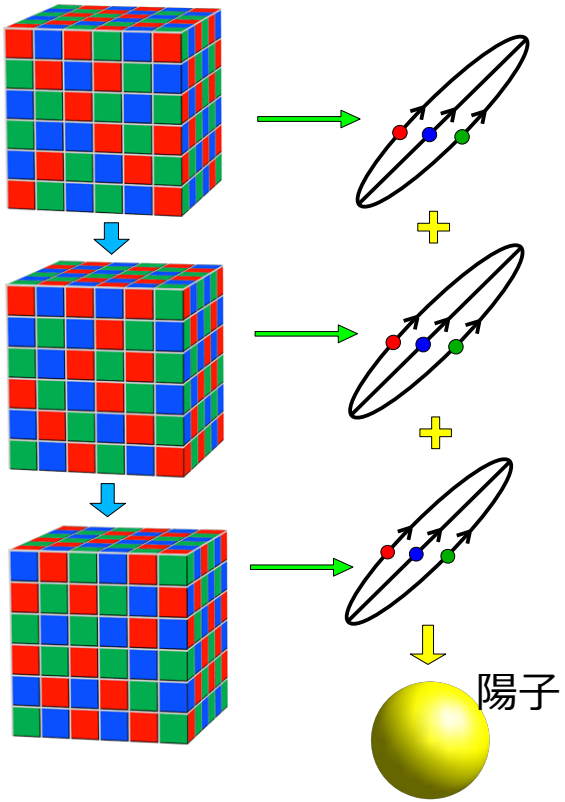
- グルーオン場の「配位」： 分子動力学的に作る
- 各ステップでクォーク場に対する線形方程式を解く必要 (大規模疎行列)

精密計算の必要性

- 生成したグルーオン場の配位を使って、物理量を計算
  - 例： クォークの伝播関数（D^{-1}）からハドロンの相関関数を構成
  - 統計平均→物理量の期待値

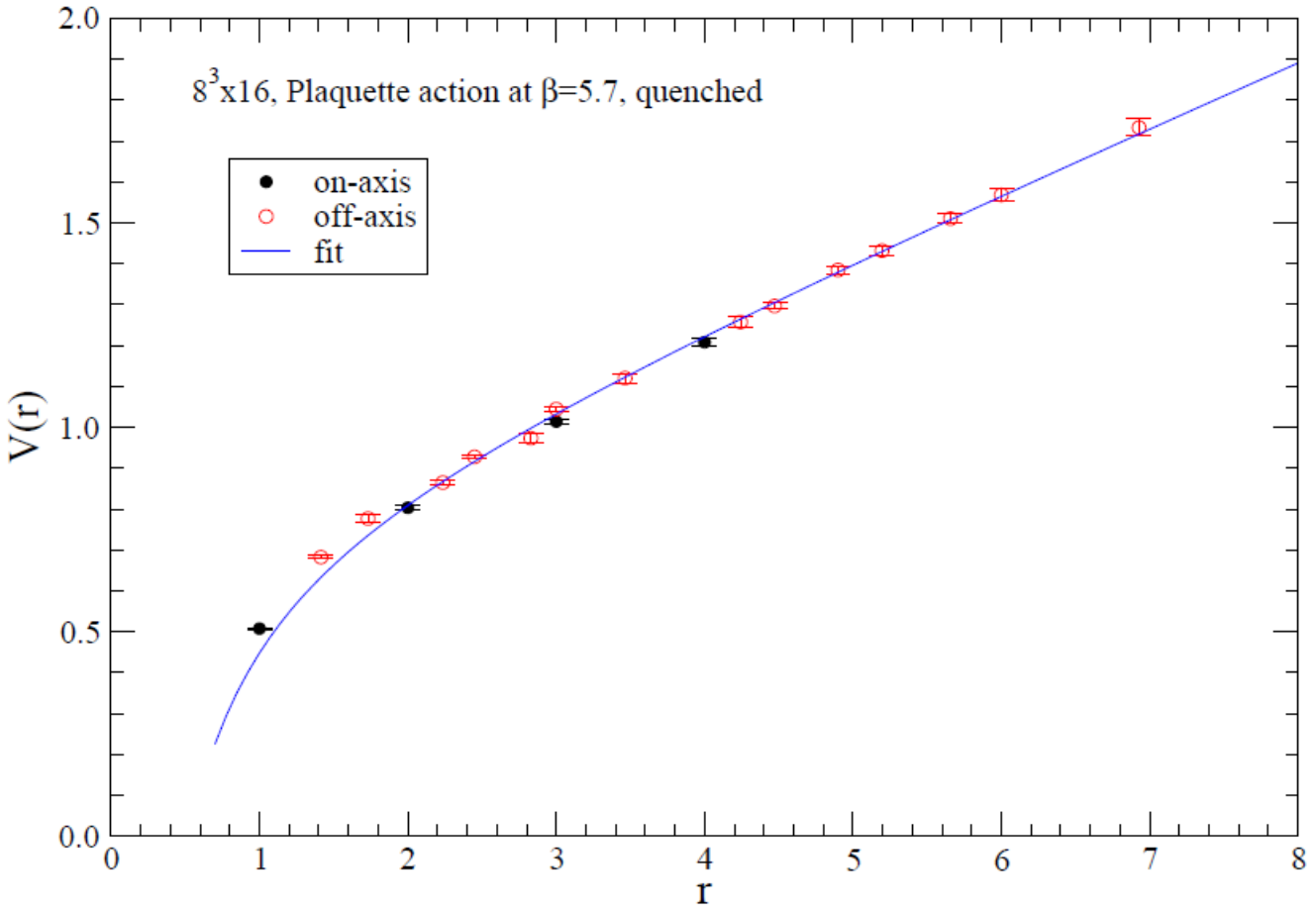
- 効果的なシミュレーションのためには、
  - グルーオン場の配位の効率的生成
  - クォークの伝搬関数の高速計算 (線形問題)
    ← 特にこれが bottleneck
  - 効率的な測定 (配位データの有効利用)

  が必要
  - スパコンでのグランドチャレンジ
  - 専用計算機の開発： QCD-PAX, CP-PACS, QCDOC, APE, etc.



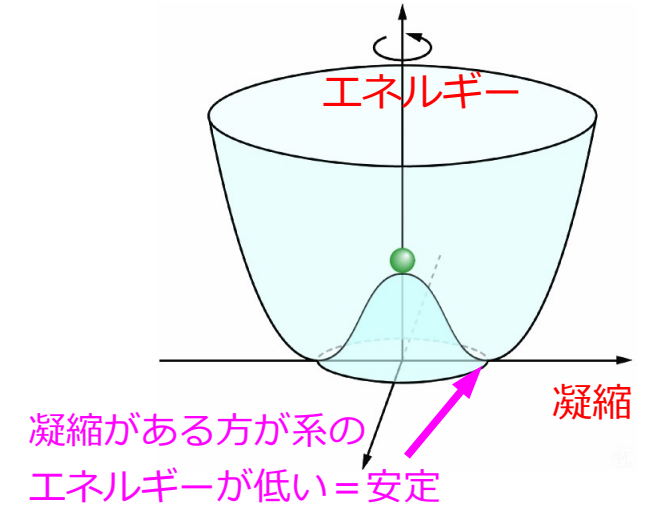
格子QCDシミュレーションの原理

- 簡単な例： クォークと反クォークの間に働く力
  - クォークと反クォークを、距離 r 離して置く
  - r を変えながら位置エネルギーを測定してゆく



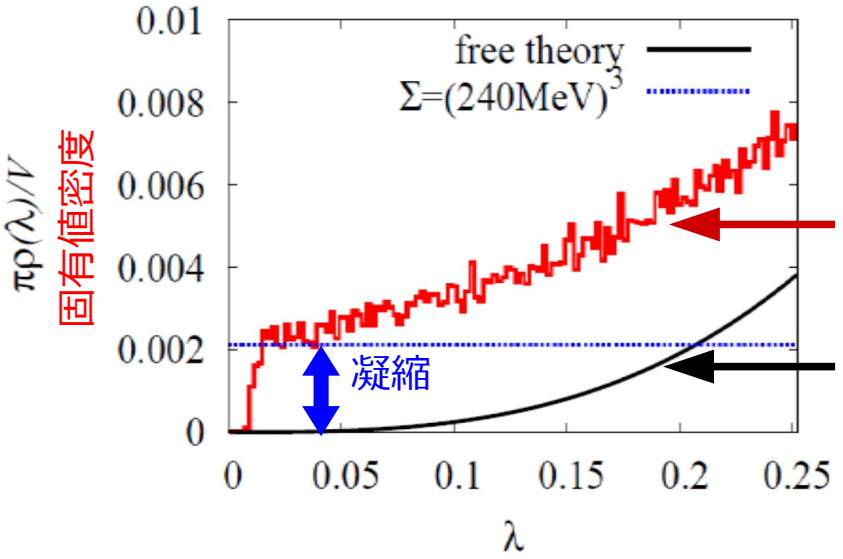
QCDの真空

- 真空は「空っぽ」ではない
  - カイラル対称性： 右回りと左回りのクォークが独立にふるまう対称性
  - 真空中にクォーク・反クォーク対が凝縮すると、対称性の破れた状態 → クォークは有効質量を獲得
    陽子、中性子の質量の 98%



- 格子QCD： クォーク・反クォークが本当に凝縮していることをQCDから直接検証
  - クォークの伝搬を表す行列の固有値密度

南部博士が提唱
2008年ノーベル賞



シミュレーションの結果

凝縮がない場合に予想される結果

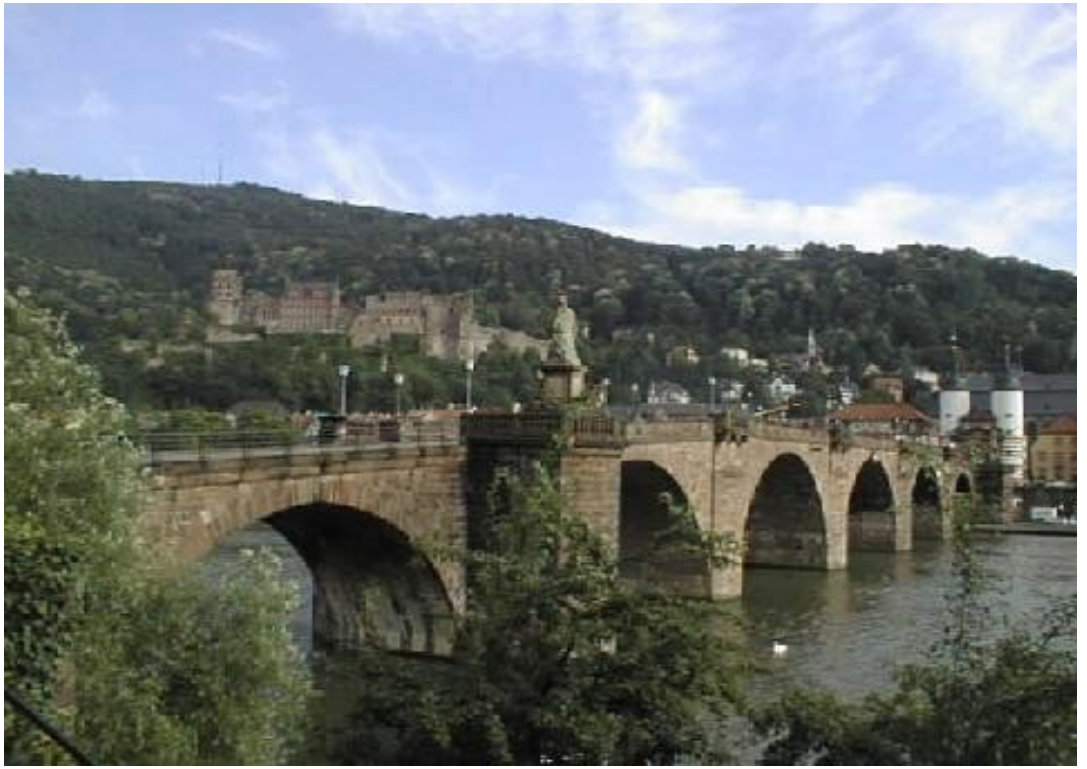
JLQCDcollaboration (Fukaya et al.),
Phys. Rev. Lett. 98, 172001 (2007)

格子シミュレーションでできること

- ハドロンとQCD真空の性質
  - 核子や中間子の質量、形状因子、磁気能率など
  - カイラル対称性とクォークの閉じ込めの理解
- ハドロン反応過程の精密計算
  - 標準理論の検証に重要
- QCDの相構造 → 宇宙初期、重イオン衝突実験との比較
  - 有限密度系の計算手法が課題
  - 最近の発展： 複素Langevin法など
- 陽子や中性子の間に働く核力の性質 （最近大きく進展中）
  - クォーク(QCD)から核力、原子核構造の理解へ
- 標準理論を超えた物理の探索
  - 新しい理論(テクニカラー、超対称性など)で何が起こるか？
  - QCD以外の理論にも同じ手法が適用されている

→ 汎用性の高いコードの必要性

汎用格子ゲージ理論コード Bridge++ の開発



汎用QCDコードの課題

- フェルミオン演算子の種類 × 改良法
  - 格子化によって破る対称性など多様性がある (自由度が異なる)
  - 格子化による誤差を減らす、様々な改良法が提案されている
    - Wilson/Clover(improved Wilson)
    - Staggered
    - Domain-wall, overlap (better chiral symmetry), etc.
- アルゴリズム
  - 配位生成 (Hybrid Monte Carlo 法)
  - 線形方程式の解法、固有値問題などのアルゴリズム
    - アルゴリズムと実装の分離が重要
    - 測定量： 相関関数、熱力学量など
- 十分な性能
  - 移植性 (プラットフォーム、コンパイラ)
  - アーキテクチャによって異なる最適化

開発の経緯

- これまでのコード開発
  - 国内では Fortran が主流： マシンごとに開発、最適化
    - 特定のプロジェクトに必要な機能を実装
    - Lattice Toolkit (Fortran) のような活動もあった
    - CCS code – performance tuning のベース
  - C/C++での公開コード (USA, Europe)
    - 読みやすさ、サポートに難あり (新しいことに使おうとすると大変)
- Bridge++開発の経緯
  - 2009年10月 新学術領域研究「素核宇宙融合による計算科学に基づいた重層的物質構造の解明」(代表・青木慎也) の一環としてスタート
  - 国内に汎用(共通)コードがないことの disadvantage
  - 2012年7月24日 最初の公開版(ver.1.0)リリース
  - 現在の最新版は ver.1.5.3 (2019年12月)

開発体制

- 現在活動中のメンバー

  赤星友太郎 (京都大), 青木慎也 (京大基研), 青山龍美 (KEK),
  金森逸作 (理研神戸 R-CCS), 金谷和至 (筑波大), 松古栄夫 (KEK),
  滑川裕介 (KEK), 根村英克 (阪大 RCNP), 谷口裕介 (筑波大)

- 活動
  - 月に1-2度のTV会議: 資料を作って回覧、wiki に議事録
  - Redmine にコードの repository
    - Subversion: check out して開発、commit して共有
    - Branch で新機能の開発 → メインコード (trunk) にマージ
  - Wiki : コードの使用法、実装のガイド、実装ノート(詳細)など
  - コードの注釈： doxygen を利用
- 今後の課題
  - 各アーキテクチャでの最適化コードの公開
  - 公開の仕方：現在はコードセット(tar.gz)をダウンロード → GitHub ?
  - ユーザサポート体制の整備 (MLなど)

Bridge++の開発方針

- 開発方針 (目標)
  - 可読性： 初心者にも使いやすく、理解可能であること
  - 拡張性： 必要な機能の追加、修正が簡単にできる
  - 移植性： 多くのプラットフォーム、コンパイラで動く
  - 高性能： 実際の研究に十分なパフォーマンス
- オブジェクト指向？
  - コード再利用性のための仕組み、テクニックが蓄積されている
    - 機能ごとにオブジェクトとして分離、再利用
    - デザイン・パターン： 典型的な実装法のカタログ化
  - Performance も必要 → プログラミング言語は C++
- 並列化
  - 分散メモリ並列化 → MPI をラップして使用、他ライブラリに換装可
  - マルチスレッド化 → OpenMP: 現在は線形アルゴリズム部分のみ

FortranからC++へ

手続き型プログラミングからオブジェクト指向プログラミングへ

- 手続き型
  - プログラミング言語： Fortran, C
  - 「データ (もの)」は配列や構造体で表し、「操作」は関数やサブルーチンで表す
- オブジェクト指向
  - プログラミング言語： C++
  - 「データ」も「操作」も、「オブジェクト」を単位として表す
  - オブジェクト間の通信によって処理を進めてゆく
- クラスを単位としたプログラミング
  - 型、構造体の一般化： オブジェクトを生成する鋳型のようなもの
  - オブジェクトが持つデータと、それを操作するメソッドを定義
  - データ、メソッドに対するアクセス権を設定 (カプセル化)
  - 継承、多態性

FortranからC++へ

- 例：格子上の場（ゲージ場、フェルミオン場）は、Bridge++ では Field というクラスのオブジェクトで表す
  - 線形代数的な処理がメソッドとして用意されている
  - 群やスピノールに特有の演算は派生クラスで実装
- Field のオブジェクトに対し、ある操作を行うオブジェクト
  - フェルミオン演算子 (行列)： Fopr
    - mult() を持つ (行列・ベクトル積に対応)
  - 線形方程式ソルバー： Solver
    - solve() を持つ (線形方程式を解いて解を返す)

  → Field (ベクトル)、Fopr (行列) を使ってアルゴリズムを構成

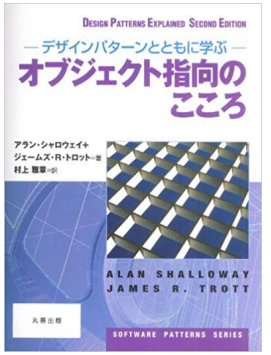
Bridge++の開発方針

- オブジェクト指向プログラミング言語に必要な条件
  - クラスを単位に機能を実装
  - カプセル化： アクセス権の制御
    - データ、メソッドに public (外部からアクセス可), private (クラス内からのみアクセス化) などを指定
  - 継承： あるクラスを拡張して別のクラスをつくる仕組み
    - 元のクラスの機能を引き継ぎながら、新しい機能を追加
    - 「基底クラス」←「派生クラス」 (図ではこの向きの矢印)
  - 多態性： 同じインターフェースをもつオブジェクトを同様に扱う
    - 同じメソッド呼び出しに対し、異なるオブジェクトが異なるふるまい
    - コンパイル時に決まる多態性を静的多態性とよぶこともある
- これらの機能を組み合わせて、どう使うかには大きな自由度有り
  - デザインとしての指針が必要
  - デザイン・パターンの活用
    - Gamma et al. (“GoF”), *Design Patterns: Elements of Reusable Object-Oriented Software*

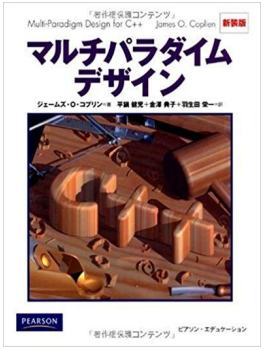
Bridge++の開発方針

個人的に参考になった本 (C++の教科書以外で)

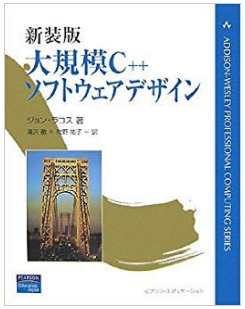
- シャロウェイ＋トロット「オブジェクト指向のこころ」
  - デザインパターンでオブジェクト指向プログラミングの原則を示す
  - 継承と多態性の使い方の指針



- コプリン「マルチパラダイムデザイン (新装版)」
  - 共通性／可変性分析
  - 多態性をどの段階（コンパイル時、実行時）で、どの方法（マクロ、template, 動的多態性）で、実現するかを分析



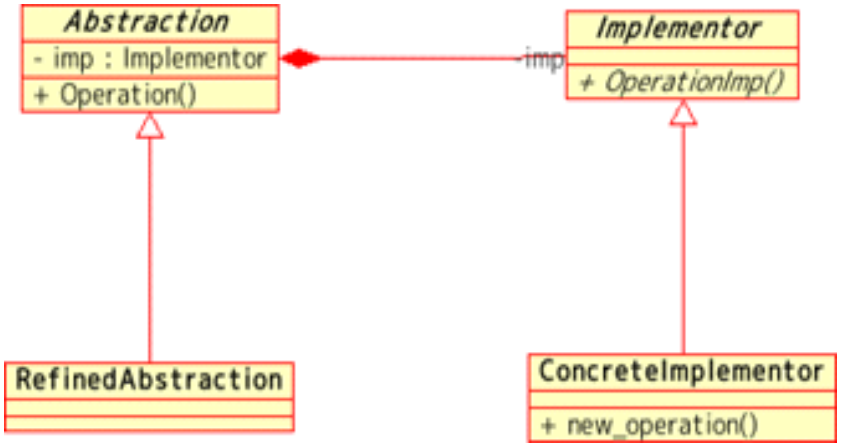
- ラコス「大規模C++ソフトウェアデザイン」
  - 大規模化してゆくコードセットの中で、各要素の独立性を高め、依存性を減らすテクニック



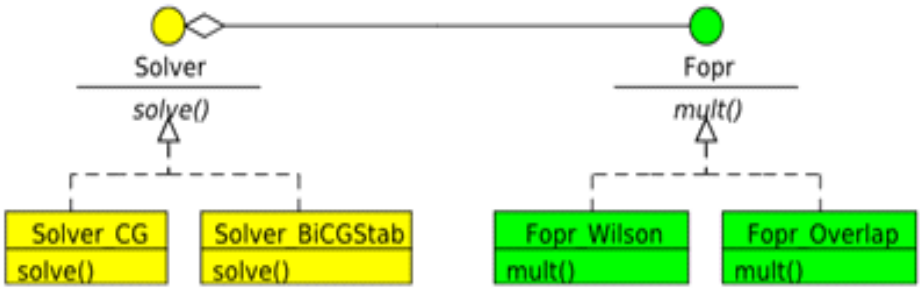
デザインパターンの例

- Bridge パターン
  - 目的: 関連の強い二つのクラスを互いに影響しないように拡張する

    日立ソフトウェアエンジニアリング「Java デザインパターン徹底攻略」
  - 適用例: 線形ソルバー (Solver) とフェルミオン演算子(Fopr)
    - Solver の呼び出し側は、線形方程式を解いてくれれば良い
    - Solver としては Fopr は mult を持っていれば良い
    - 基底クラス (Fopr) にmult の作法を記述しておき、サブクラスに具体的なフェルミオン演算子を実装する (多態性)



BridgeパターンのUML図



Bridge++でのクラス構造

Bridge++のデザイン

Bridge++はどの程度オブジェクト指向か？

- 実装レベルではかなり手続き型スタイル
  - やはり Fortran-like な方が人にもコンパイラにも理解しやすい
  - 式(expression)テンプレートは不使用
  - クラス外の関数もある
- 機能ごとにまとめるレベルでオブジェクト指向を意識
  - デザインパターンを参考に
  - 継承は多態性と組み合わせて使う場合に限って利用
    - E.g. Clover フェルミオンを Wilson フェルミオンから派生させない
- 個人的には「オブジェクト指向の皮を被ったFortran」が理想
  - ではあるが、C++の表現力は利用したい

C++での開発

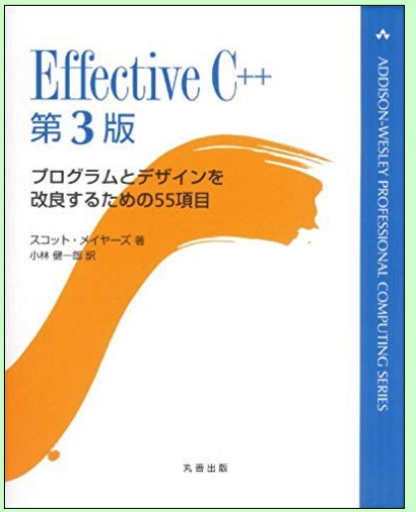
- C++ での難しさ： できることが多すぎる

スコット・メイヤーズ「Effective C++ 第3版」

1項 C++を複数の言語の連合と見なそう

- C
- オブジェクト指向C++
- テンプレートC++
- STL (Standard Template Library)

「C++で効率よくプログラミングするためのルールは、
C++のどの部分を使うかで変わってくる」



- コーディングスタイルの確立と共有が重要

← その試行錯誤がこれまでのBridge++開発の歴史

- 「スタンダード」を策定

開発上の決断

- 例外はどうするか？ → 原則として使わない
  - コードの複雑化を避ける
  - 問題 (overflow や zero division)が起きた時は潔く止まっても良い
- C++11 に準拠するか？ → 準拠しない
  - Unique pointer などどうしても使いたい機能は別実装も用意
- 複素数は std::complex ？ → C スタイルも可
  - C の代替実装も用意、どちらでも動く関数を定義してそれを利用
  - 一部プラットフォームでのパフォーマンス向上のため
- 並列化 (分散メモリ)は？ → MPI をラップして使用
  - 低レベル通信ライブラリで置き換えることも可
- 並列化 (マルチスレッド)は？ → OpenMP
  - Pthread も検討したが、コードが複雑になりそう
  - OpenMP なら実装できる人が多い
- パラメーターファイルのフォーマット → YAML が基本
  - XML も使える (YAML → XML コンバーターも実装)

可読性

- 読みやすいコードとは？
  - E.g. axpy: y = y + a*x
  - 式 (expression) template を使うか？
    - C++ らしい？書き方
    - 数式的な可読性
  - BLAS like な関数で処理するか？
    - Fortran-like なアプローチ
    - 計算機がやっている処理としての可読性
  - BLAS-like approach を採用： axpy(y, a, x) のように書く
    - 式 template は使いこなすのが難しい → 想定外の事態に対応が難しい
    - Multi-thread 化などが容易
- 配列のスタイル
  - メモリ領域を確保 (STL のコンテナもしくはポインタ)
  - a[k][j][i] ではなく、a[index(i, j, k)] のような書き方
    index() はインライン関数

拡張性と再利用性

- 再利用性
  - 線形ソルバーなどのアルゴリズムを一度実装すれば、どのような行列(フェルミオン演算子)にも使える
  - これは Fortran で使われてきた、BLAS/LAPACK でも実現
  - オブジェクト指向は再利用可能な部品をどう作っていくかという考察から生まれたので、自然な形で実現できる
- 設計の際に注意すべきはインターフェースの仕様
  - 基底クラスでインターフェースを定義
  - 派生クラスで実装
  - 派生クラスを追加することで新しい機能を追加可能
- 全く新しいタイプの機能を追加したいとき
  - デザインのスタイルを踏襲して基底クラスを追加

移植性

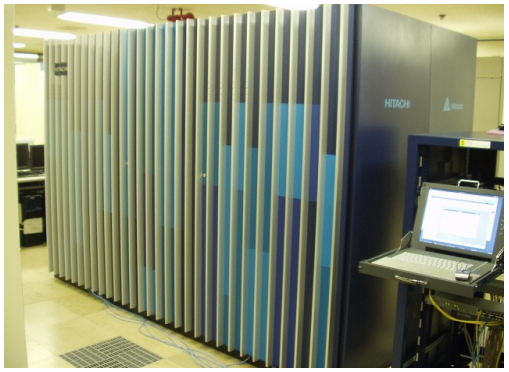
- できるだけ多くのプラットフォームでコンパイル、動作確認
  - システム
    - PC, Xeon Cluster
    - Oakforest-PACS (Intel Xeon Phi)
    - NEC SX-ACE, SX-Aurora (vector)
    - Fujitsu FX100 (SPARC)
    - 以前は IBM Blue Gene, Hitachi SR16000 なども
  - コンパイラ
    - GNU, Intel, PGI, Cray, Fujitsu, NEC SX
  - プラットフォーム依存がある場合
    - define macro で個別に回避
    - コンパイル可、動作確認済の条件でのみ make 可にする

最新アーキテクチャへの対応

葛飾北斎 冨嶽三十六景《深川万年橋下》
Hokusai, “Fukagawa Mannen-bashi shita” in Fugaku Thirty-six scenery

最新アーキテクチャへの対応

- Bridge++ の制限
  - データレイアウトが固定
  - 倍精度実数
- これらは開発当初、早く動くコードが必要だったための選択
  - 当時使っていたシステムの特徴に依存
    → 実際に使いながら、デザインに必要な条件の洗い出し
- 最新のアーキテクチャに対応するために構造的に拡張中



これまでの活動

- Blue Gene/L, Q
  - 開発開始当初の最適化対象： 典型的な超並列コンピュータ
  - スレッド毎に通信可能な低レベル通信ライブラリ
  - ここまではクラスの置き換えで対応
- GPU, Pezy-SC
  - 拡張の仕方を模索し始めた
  - 演算加速部へのオフロード： OpenACC, OpenCL による実装
- Intel Knights Landing (AVX-512)
  - SIMD アーキテクチャへの対応
- SX-Aurora TSUBASA (in progress)
  - ベクトルアーキテクチャへの対応
- Fugakuに向けての準備 (in progress)

Extended Bridge++

Extended Bridge++ framework:

Core library + *extension (“alternative”)*

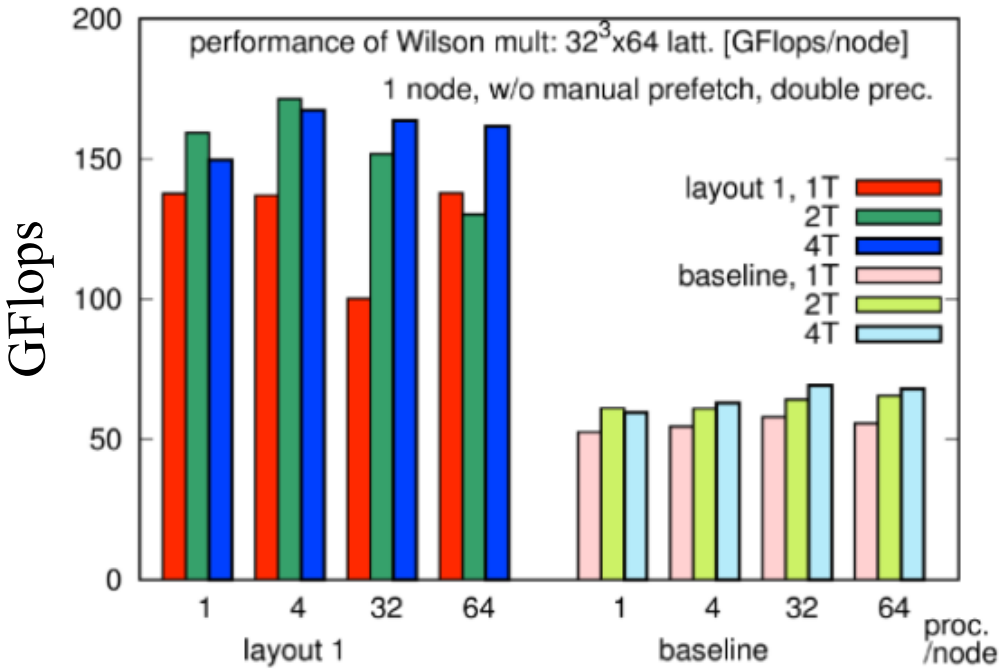
- Bridge++ core library
  - これまでの Bridge++ コード (開発はまだ続いている)
  - 枠組み、汎用ツールを提供
  - 最適化が不要な部分
- 拡張 (“alternative” コード)
  - 任意のデータレイアウト、実数型
  - Core library と同じクラス構造で C++ テンプレートを利用
  - 各アーキテクチャに特化した最適化を組み込む
  - 現在は未公開、直接リクエストに応じて提供
- 場のオブジェクト
  - `Field → AField<REALTYPE, IMPL>` (IMPL is dfnied by enum)
    - e.g. `AField<double, SIMD>`

SIMD architecture

- SIMD アーキテクチャ(Intel AVX-512 on Xeon Phi KNL and Xeon)

  I.Kanamori and H.Matsufuru, LNCS 10962 (2018) 456

  - Oakforest-PACS (JCMCIA) を利用
    - Peak 6(SP)/3(DP) TFlops/node
  - データレイアウトの変更 → SIMD ベクトルへの適用
  - AVX-512 intrinsics とマニュアル・プリフェッチによる最適化
  - Bridge++ core library との比較: 2-3 倍性能向上



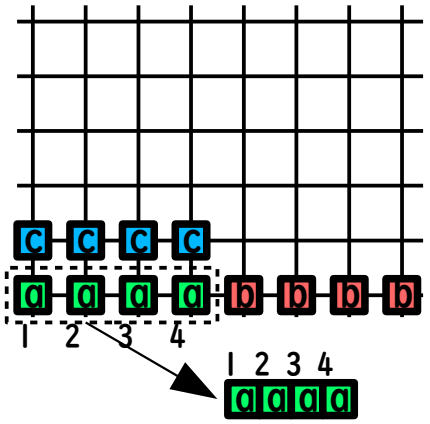
Oakforest-PACS

SIMD architecture

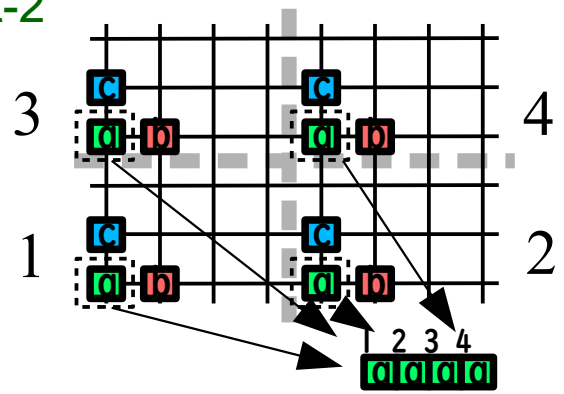
- データレイアウトによる性能の違い
  - シンプルなレイアウト (layout-1)
  - "Grid"-type (layout-2) P.Boyle et al. PoS (LATTICE2015) 023

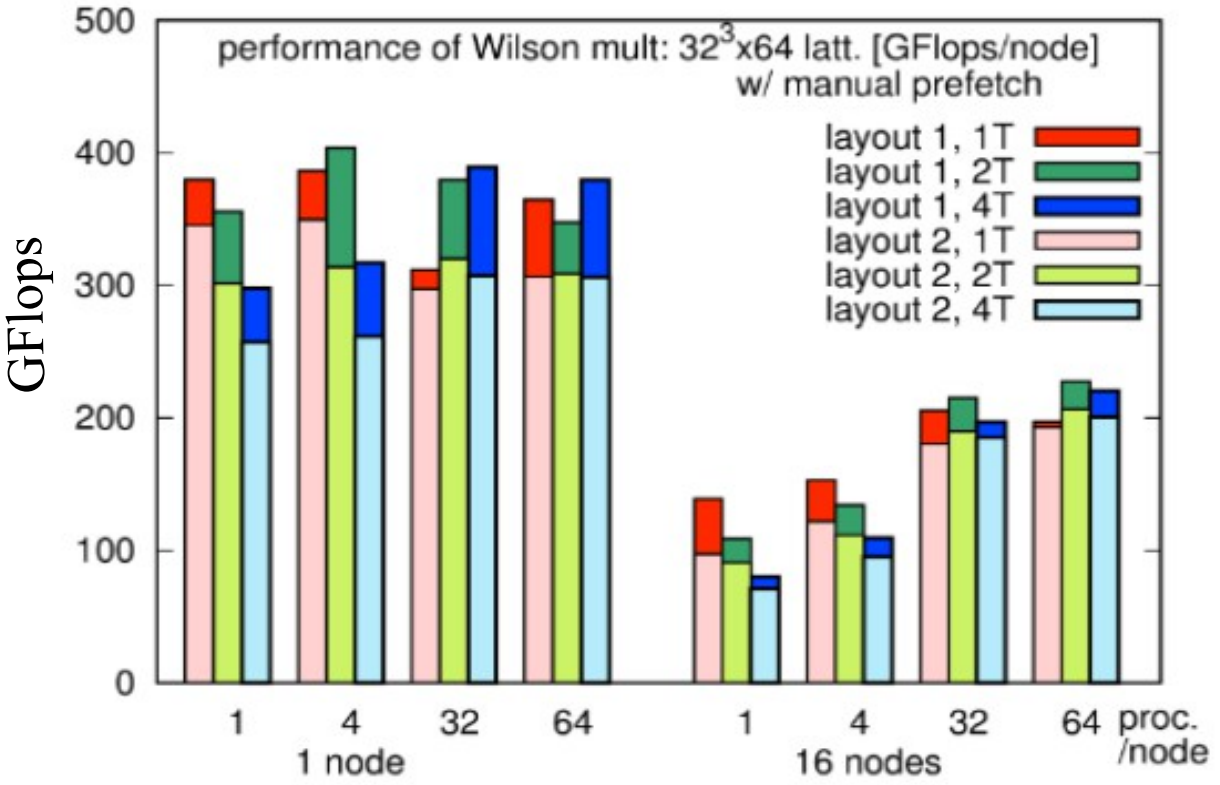
→ layout-1 が 15%程度高速だった (大きな差はない)

layout-1



layout-2



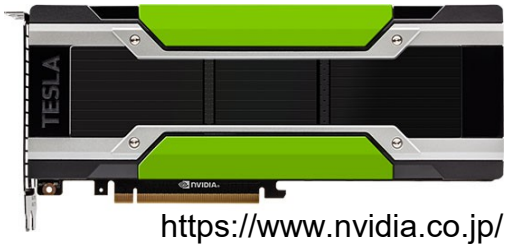


GPU, Accelerator

GPU, 演算アクセラレータ

- 線形方程式解法のオフロード

- GPU (NVIDIA, AMD)
  - OpenCL and OpenACC
    - S.Motoki et al., Proc. Comp. Sci. 29 (2014) 1701
    - H.Matsufuru et al., Proc. Comp. Sci. 51 (2015) 1313
- PEZY-SC
  - OpenCL (PZCL)
    - T. Aoyama et al. Proc. Comp. Sci. 80 (2016) 1418



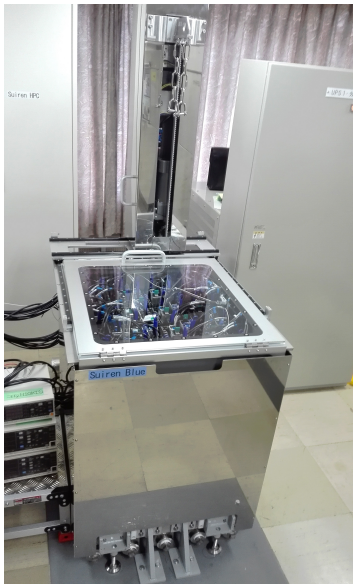
https://www.nvidia.co.jp/

NVIDIA P100

オフロードを組み込むコードデザイン

- 拡張コードの枠組で対応可能
- 実装は OpenACC を基本に
- SIMD と同時利用できるように調整中

PEZY-SC



Suiren Blue

Fugaku

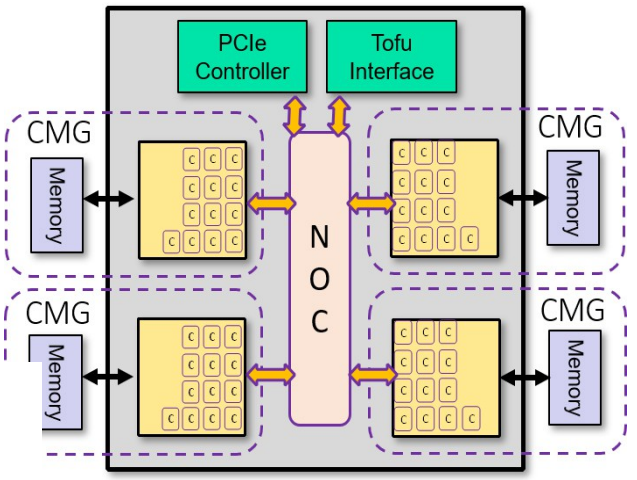
Fugaku will be available in 2021

- ARM based processor by Fujitsu with Scalable Vector Extension
  - Architecture: Armv8.2-A SVE 512bit
  - Core: 4 CMGs (NUMA nodes), 48 cores for compute
    DP: 2.7+ TF, SP: 5.4+ TF, HP: 10.8 TF
  - Cache: L1D/core: 64 KiB, L2/CMG: 8 MiB, 16way
  - Memory: HBM2 32 GiB, 1024 GB/s
  - Tofu Interconnect D (28 Gbps x 2 lane x 10 port)

(From Fujitsu site)

- Prototype was aworded top ranking of Green500 in Nov. 2019
- RIKEN Fugaku processor simulator is available for application development

DP 3.072 TF/core
3.379 TF (boost)
(From Kodama-san's talk at QCD coding workshop Dec 2019)



(From RIKEN R-CCS site)

Preparation to Fugaku

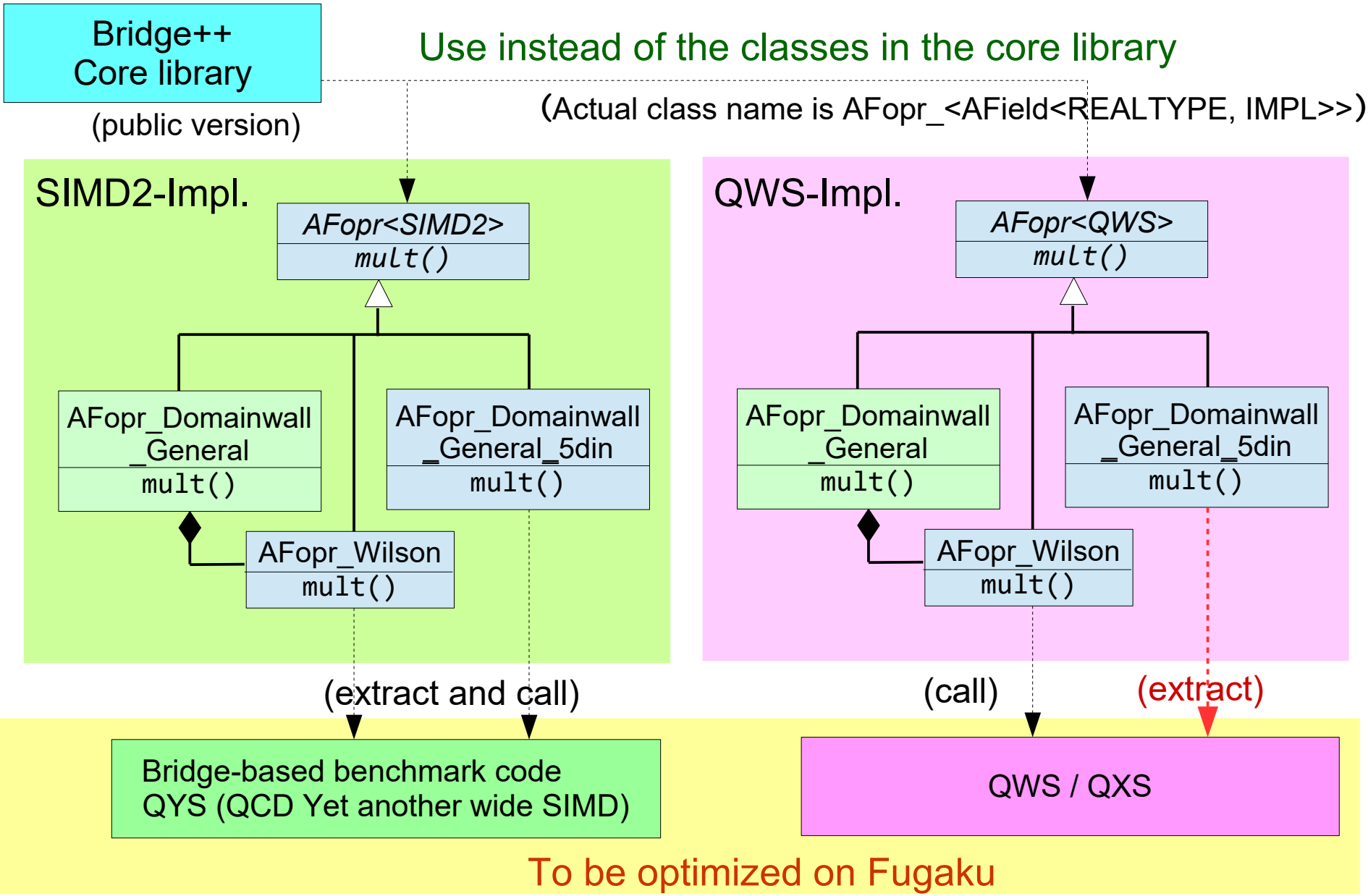
Basic strategy

- 出来るだけ *QWS* を利用
  - “QCD Wide SIMD” library (R-CCS/FS2020で中村さん中心に開発)
  - Available: (domain-decomposed) even-odd clover fermion
- *Bridge++* での実装
  - フェルミオン演算子：Domainwall, Staggered, etc.
  - QWS の最適化テクニックを適用

準備

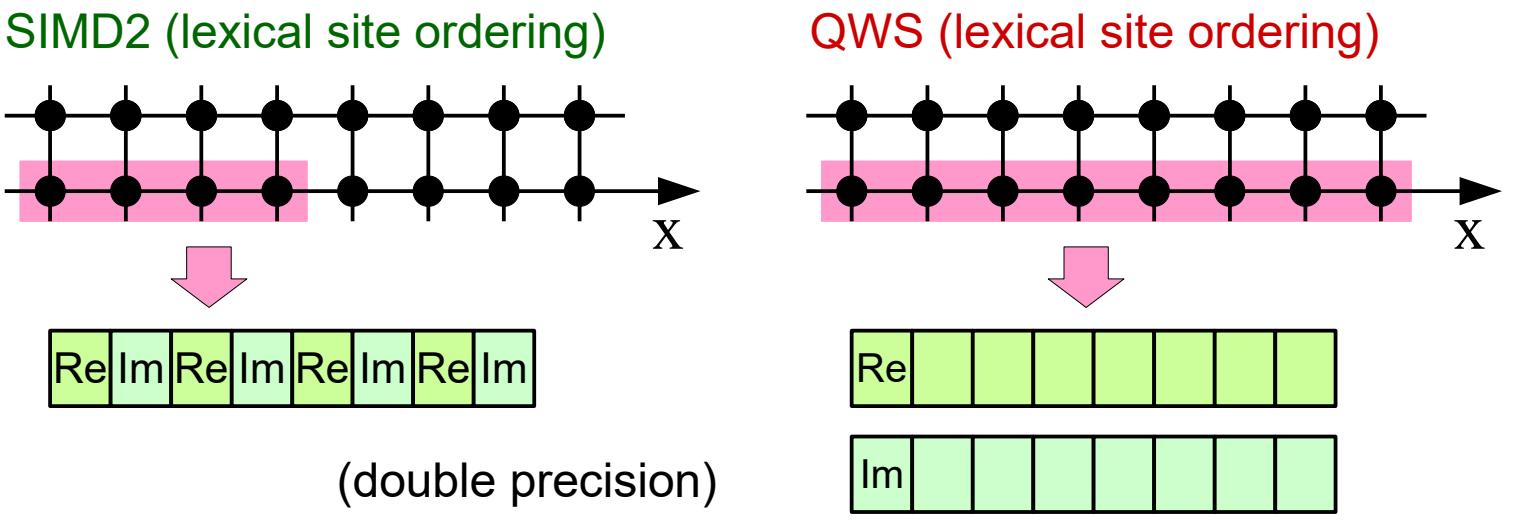
- “QWS” implementation
  - QWS と同じデータレイアウト、コンベンションでのブランチを準備
  - QWS を呼ぶ前後でデータの変換
- “SIMD2” implementation
  - SIMD (for AVX-512) アーキテクチャのために開発したものを利用
  - Fugakuでも性能が出る可能性 → 最適化して比較の予定

Preparation to Fugaku



Preparation to Fugaku

- Length of SIMD vector
  - `VLEND=8 (double), VLENS=16 (float)`
  - Restriction of local lattice size in x-direction
- SIMD2 impl.
  - VLEN/2 sites in x-direction are packed in a SIMD vector
- QWS impl.
  - VLEN sites in x-direction are packed in a SIMD vector



まとめ

汎用格子ゲージ理論コード Bridge++ の開発

- 格子QCDシミュレーション
- プログラム開発の視点から
  - オブジェクト指向と C++ での実装
  - 可読性、拡張性に関する議論
- 最新アーキテクチャでの最適化に向けた、デザインの拡張
  - SIMD アーキテクチャ向けの実装
  - GPU, PEZY-SC: 演算アクセラレータへのオフロード
  - Fugaku に向けての準備

新しいアイデアや最新のアーキテクチャを試すのに、汎用コードは有用