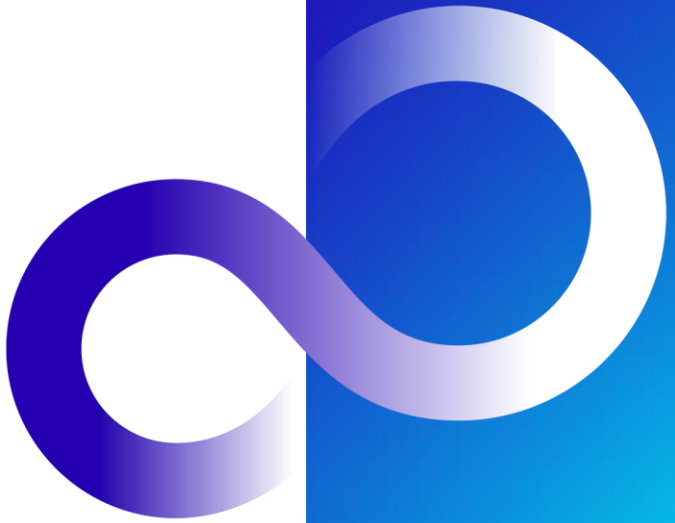
OpenMPによるA64FX並列プログラミング入門

2024.2.5

富士通株式会社



目次

13:00 - 13:30	スパコンの使い方など
13:30 - 14:20	A64FXアーキテクチャと並列プログラミング（座学） 前処理付き勾配法の説明
14:30 - 15:20	OpenMPを活用した並列プログラミング演習（前編） 環境へのログイン プログラム確認 CSRフォーマットでのPCG法の並列化
15:30 - 17:00	OpenMPを活用した並列プログラミング演習（後編） ELLフォーマットでのPCG法の並列化 更なる最適化

FUJITSU

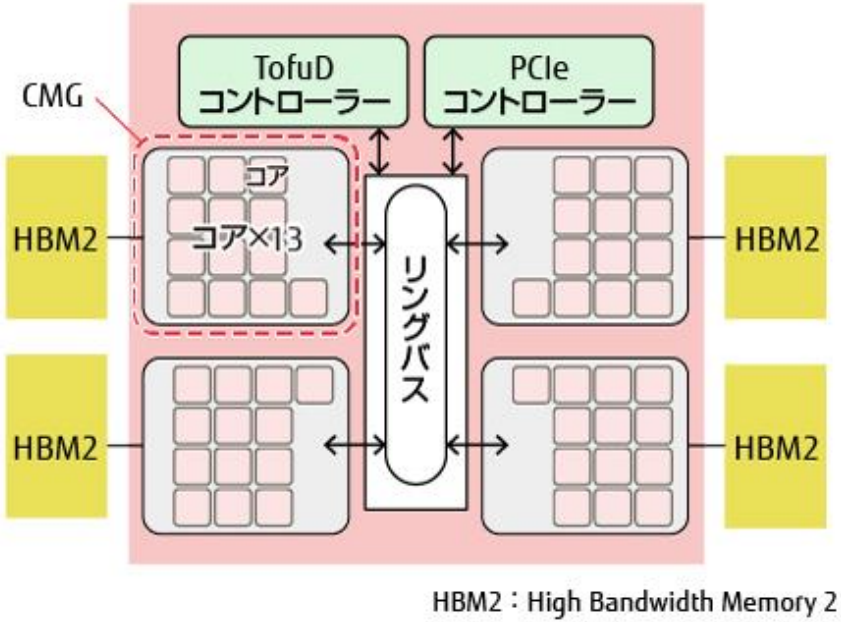
A64FXアーキテクチャと並列プログラミング

A64FXアーキテクチャ

A64FXの概要

- 富士通が2019年にリリースしたメニーコアCPU
- 48 (+アシスタント)コア、コアあたり512bits SIMD演算器 x2
- 高バンド幅のメモリを搭載
- 2023年6月第2位の「富岳」に使われるCPU
- 2019年にはGPUのシステムを抑えてGreen500で1位を取るなど、非常に電力効率がよい

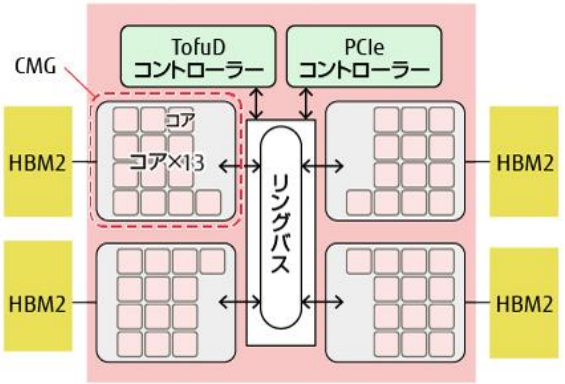
A64FX CPU



https://www.fujitsu.com/jp/about/resources/publications/technicalreview/2020-03/article03.html

A64FXのハードウェアの特徴

- 汎用CPUとしては非常に高い電力性能
  - 高い理論演算性能
  - 高いメモリバンド幅
- 一方で使いづらいところがある
  - メモリ容量が小さい（32GiB）
  - 命令レイテンシが長い（FMAは9cycles）
  - ラストレベルキャッシュメモリサイズが小さい
  - すべてのコアとSIMD演算器を使わないと高性能は達成できない



HBM2：High Bandwidth Memory 2

コア数	48 + アシスタントコア2~4
周波数	2.0GHz (or 2.2GHz)
理論演算性能	3.4 TFLOPS
メモリ容量	32 GiB
メモリバンド幅	1024GB/s
L1 Cache	64 KiB/core (Inst/Data)
L2 Cache	8 MiB / CMG

A64FXのソフトウェアから見た特徴

FUJITSU

- 得意なプログラムをチューニングすれば非常に高性能
  - 例1：最適化されたライブラリが主体のプログラム
  - 例2：メモリバンド幅ネックかつ並列性が十分にあるプログラム
  - 例3：演算ネックで、並列性があり、レイテンシを隠蔽できるプログラム
- チューニングなしでは性能が出にくい
  - 既存CPU向けに記述されたコードでは性能が活かせないことが多い（コアとSIMDの並列性の抽出が必須。演算ネックのプログラムだとレイテンシの隠蔽も必要）
  - アルゴリズムとデータ構造の見直しも必要かもしれない
- 苦手なプログラムは諦めたほうがいい（こともある）
  - 演算間の依存関係が強いループボディの大きいプログラムは性能が出しづらい

FUJITSU

A64FXアーキテクチャと並列プログラミング

A64FXの性能の引き出し方

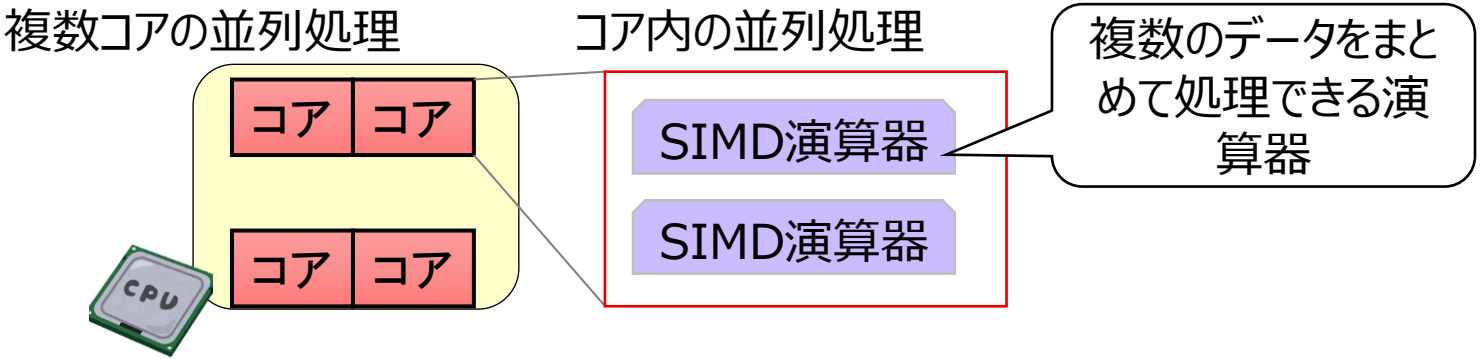
A64FXに最適化された計算ライブラリの利用

FUJITSU

- A64FXのHW性能を十分引き出した計算ライブラリが公開されている
- まずは計算ライブラリの活用を検討する
- 有名な計算ライブラリ
  - SSL2：富士通が最適化したHPC向けライブラリ（行列演算系などがある）。マルチコアで使用するときは、富士通のOpenMPライブラリを使う必要がある
  - OneDNN: Intel発のDeep learning向けライブラリ。主に、富士通とサイボウズ・ラボの光成さんでA64FX向け最適化を行い、本家OSSにアップストリームした
  - その他
    - BLAS, LAPACK, FFTなど：A64FX向けに最適化されていないものでも実行効率30-50%程度はあると思うので活用したほうが良いと思います

並列処理は必須

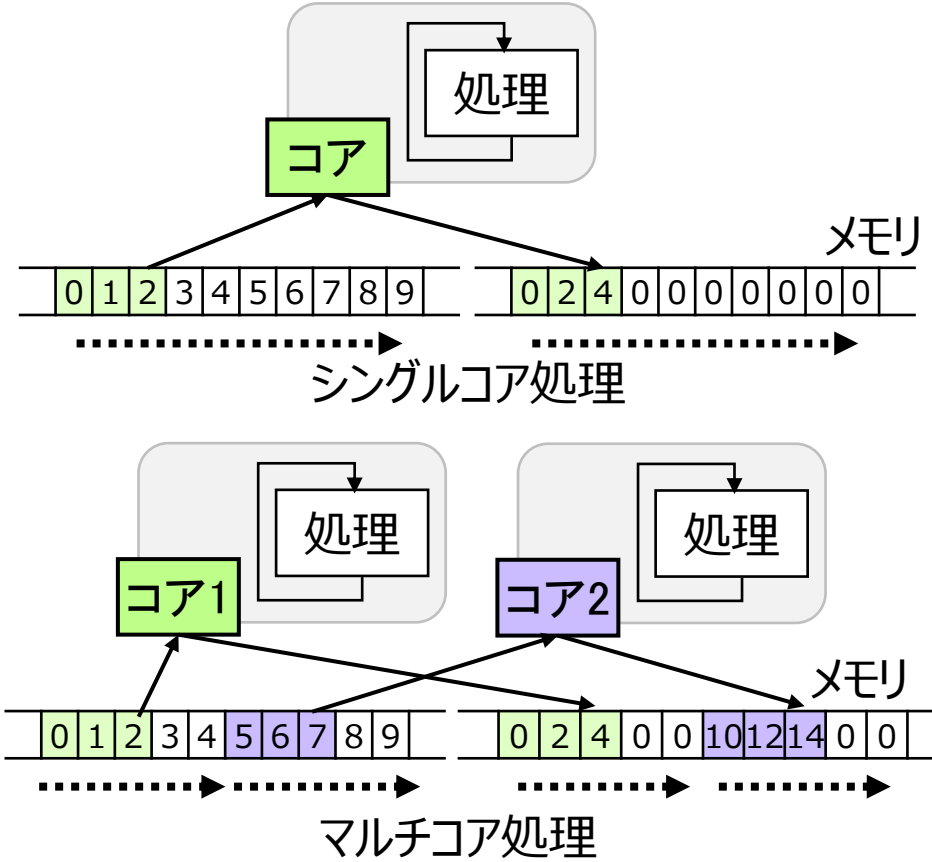
- 現代のプロセッサは、並列性を活かさなければ性能がでない



- 複数コアの並列処理とSIMD演算器の並列処理は、コンパイラによる最適化があまり期待できないので、プログラマが記述しなければならない
  - プログラムから、依存関係のない（＝並列処理可能）処理を抽出する必要がある

データ並列とスレッド並列

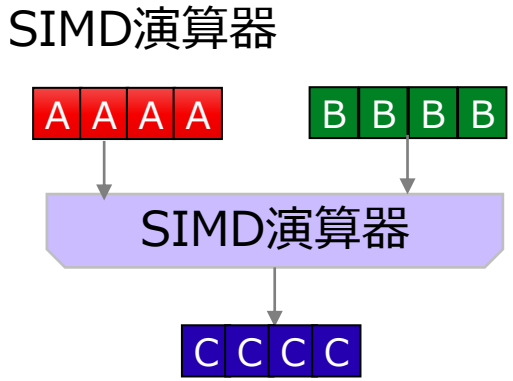
- データ並列
  - 最もよく使われる並列性
  - 異なるデータに対して同じ処理を行う
  - コア間の並列処理でもSIMD演算器の並列処理でも利用可能
- スレッド並列
  - コア間の並列処理に使われる並列性（SIMD演算器では利用できない）
  - コア間で同じコードを処理させる
- ~~タスク並列~~
  - ~~ややこしいのでなるべく考えない~~



SIMD演算器での並列処理の記述

FUJITSU

- メモリ上で連続したデータをまとめて演算する
- 最も内側のループで、連続しているデータに対して処理するように記述されていればコンパイラがSIMD演算器を活用するコードを生成してくれる



SIMD化されるコード

```
for (i = 0; i<N; i++)
 c[i] = a[i] + b[i];
```

多分SIMD化されないコード

```
for (i = 0; i<N; i++)
 c[i] = a[2*i] + b[2*i];
```

並列性の抽出

- すべてのコアを使う（A64FXはコア数が多いので重要）
  - ループにOpenMPディレクティブを挿入
  - ループの回転数を確認して足りない場合は工夫する
    - OpenMPのcollapse節を使って、多重ループに対して並列化を行う
    - 処理方法を見直す
    - プロセス並列やタスク並列の導入を検討する（今回の演習の対象外）
- SIMD演算器を活用する（A64FXではメモリバンド幅ネックのプログラムでもSIMD演算器の活用は必須）
  - 最も内側のループで以下を満たせるようにする
    - 連続データに対して処理できるようにする
    - 十分な回転数があり、なるべく512bitsの倍数でデータが処理できるようにする
    - if文はなるべくループの外に追い出す
    - 関数呼び出しは避ける（コンパイラによる最適化が難しくなる）
  - もしくは、アセンブリコードレベルで記述する

FUJITSU

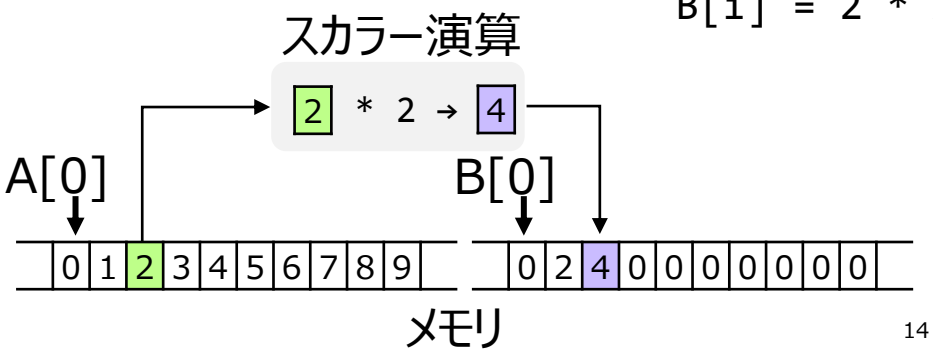
A64FXアーキテクチャと並列プログラミング

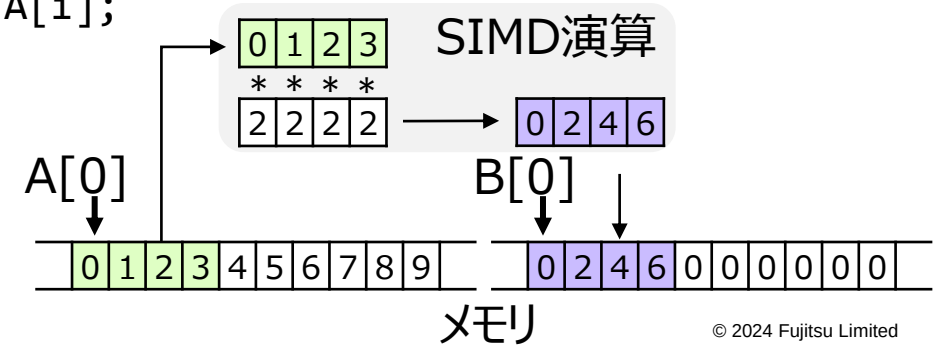
SIMD

Single Instruction Multiple Data (SIMD)

- 1命令を複数のデータに同時に適用する機構
  - 複数データを一度に処理することで演算性能を向上
- 近年のCPUの多くはSIMD機構を搭載
  - x86: SSE, AVX2, AVX512
  - ARM: Neon, **Scalable Vector Extensions (SVE)**


```
for(i=0;i<N;i++)
  B[i] = 2 * A[i];
```

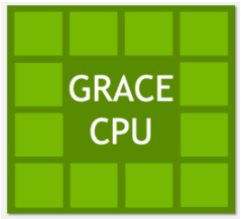




Scalable Vector Extensions (SVE)

- ARMアーキテクチャの拡張SIMD命令セット
- Vector Length Agnostic (VLA)
  - SVE命令はベクトル長として128-bit ~ 2048-bitまで128-bitの倍数をサポート
    - ハードウェアは任意のベクトル長を実装することが可能
  - ベクトル長に依存しないプログラムを記述可能

  → 汎用性がある高速なプログラムの作成を可能に



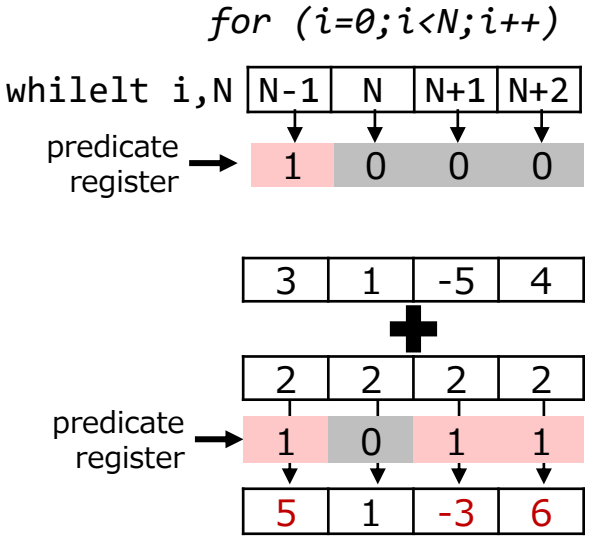
Grace CPU
128-bit SVE

Graviton3
256-bit SVE

A64FX
512-bit SVE

SVEの特徴

- プレディケーション
  - SVEレジスタの要素ごとに命令の適用の有無を変えられる
  - 多くのSVE命令がプレディケートレジスタ付きの命令
  - 利点
    - → 任意長のループのSIMD化が容易 (端数処理)
    - → if文を含む処理のSIMD化が可能



A64FXのSVE演算部

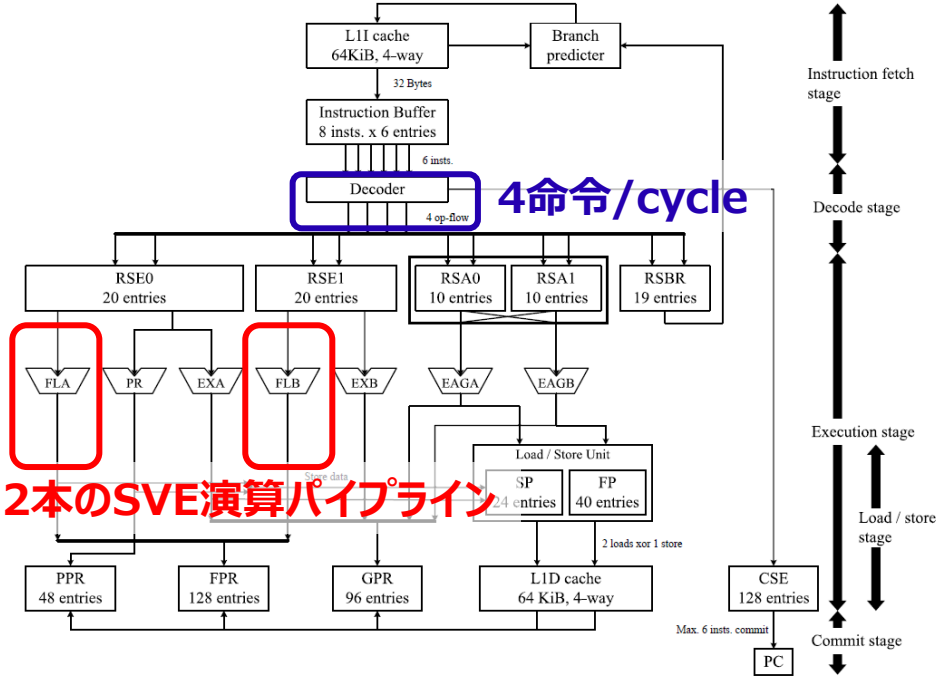
- A64FXの各コアに、2本のSVE演算パイプライン
- SVE幅: 512-bit
  - FP64: 8要素分
  - FP32: 16要素分
  - FP16: 32要素分

1度にまとめて
処理可能

- 命令発行部
  - 最大4命令発行/cycle

アプリ性能を高めるためには

SVE命令x2/cycleを目指して
SVE命令の割合が高いプログラムを作る



A64FXのコア内のプロセッシングステージ

SIMD演算器の活用

FUJITSU

- 最も内側のループでメモリ上の連続したデータに対して処理したい

```
for(i=0;i<M;i++){
  for(j=0;j<N;j++){
   for(k=0;k<K;k++){
     c[i][j]+=a[i][k]*b[k][j];
   }}}
```

- 配列aはk方向で連続アクセスとなっているのでOK
- 配列b,cはk方向で連続アクセスとなっていないのNG

- j方向のループを一番内側にする

```
for(i=0;i<M;i++){
  for(k=0;k<K;k++){
    for(j=0;j<N;j++){
      c[i][j]+=a[i][k]*b[k][j];
   }}}
```

- 配列b,cはj方向で連続となったためOK
- 配列aは最も内側のループでは同じデータを処理し続ける（おそらくSIMD化はされない）

FUJITSU

A64FXアーキテクチャと並列プログラミング

すべてのコアを使う

複数コアを使った並列処理の記述方法

FUJITSU

- OpenMPで記述
  - サーバ内（共有メモリ）での並列処理を記述するときに使う
  - 逐次プログラムにディレクティブと呼ばれる指示文を入れることで並列化する
  - 共有メモリが前提のため、明示的なデータのやり取りは記述しなくて良い
  - 記述が簡単で、コードの可読性があまり落ちないのでよく使われる

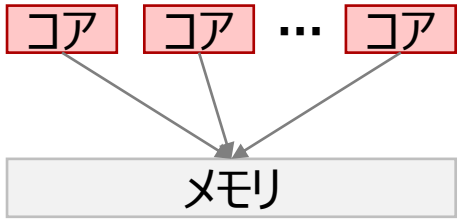
本講習の対象

- MPIで記述
  - 主に複数サーバ（分散メモリ）での並列処理を記述するときに使う
  - 共有メモリがない前提で、明示的にデータのやりとりを記述する
  - 複数サーバでの並列処理を使うときは、ほぼ必須なのでOpenMPを使わずにMPIだけ使うコードもある
- 他の言語で記述
  - Pthread, OpenCL, SYCLなどがある
  - 興味のある方は調べてみると良いと思います

OpenMPとは

- 共有メモリ型の計算機において並列処理を可能とするAPI
- 逐次プログラムにディレクティブという指示文を挿入することで並列化可能
- OpenMPディレクティブは、コンパイルオプションを指定しない限り、無視されるため同じコードで逐次プログラムと並列プログラムを生成できる

共有メモリ型計算機
＝同じデータにアクセス可能

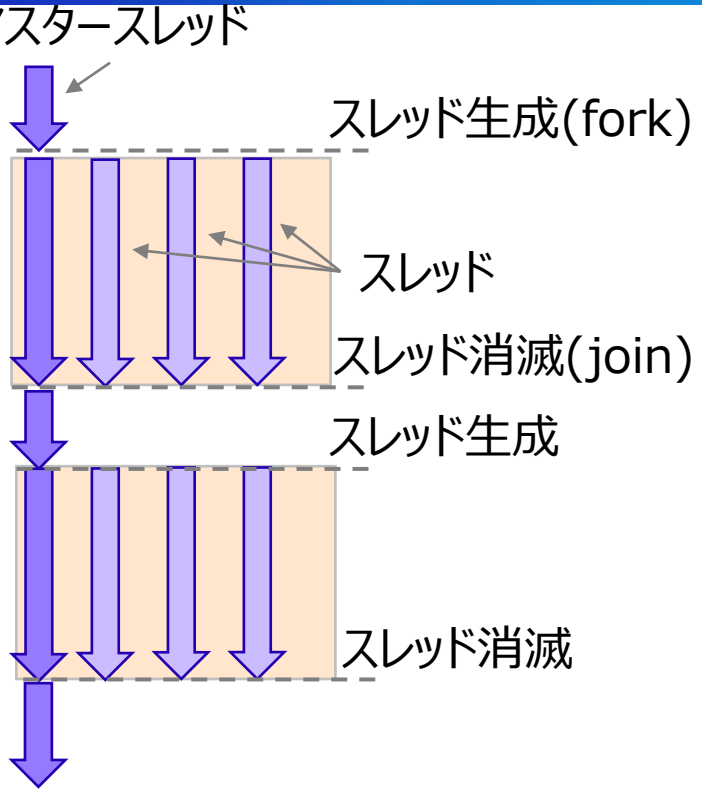


OpenMPプログラム例

```
#pragma omp parallel for
for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
```

OpenMPでよく記述される並列モデル

```
// 逐次処理
…
#pragma omp parallel
{
   // 並列処理A
   …
}
// 逐次処理
…
#pragma omp parallel
{
  // 並列処理B
  …
}
// 逐次処理
…
```



OpenMPでよく使われる並列化

FUJITSU

- #pragma omp parallel
  - スレッドを立ち上げて、並列に実行することが可能

逐次コード

```
printf(“Hello world¥n”);
// 1回だけHello worldが出力
```

並列化コード

```
#pragma omp parallel
{
    printf(“Hello world¥n”);
}
// スレッド数分のHello worldが出力される
```

OpenMPでよく使われる並列化

- #pragma omp parallel for
  - ループを対象とした並列処理
  - 変数iはデフォルトで各スレッドprivateなものとみなされる

逐次コード

```
for (i = 0;i < N; i++) {
    c[i] = a[i] + b[i];
}
```

並列化コード

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

OpenMPでよく使われる並列化

- #pragma omp ... private(...)
  - 変数が各スレッドごとのものなのか、共有なのかを明示
    - デフォルトではshared (共有) となる
    - 例にある変数 iはデフォルトで各スレッドprivateなものとみなされる

逐次コード

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    c[i * N + j] = a[i * N + j] + b[i * N + j];
  }
}
```

並列化コード

```
#pragma omp parallel for private(j)
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    c[i * N + j] = a[i * N + j] + b[i * N + j];
  }
}
// もしくは
#pragma omp parallel for
for (i = 0; i < N; i++) {
  int j; // 変数のスコープを狭める
  for (j = 0; j < N; j++) {
    c[i * N + j] = a[i * N + j] + b[i * N + j];
  }
}
```

OpenMPでよく使われる並列化

- #pragma omp ... reduction (operator: variable)
  - 内積計算など、結果を足しこみ、単一の結果（スカラー値）を得たい場合などに使用
    - 内部的には、スレッド並列にて効率的に実行される

逐次コード

```
int dotp = 0;
for (i = 0; i < N; i++) {
    dotp += a[i] * b[i]
}
```

並列化コード

```
int dotp = 0;
#pragma omp parallel for reduction (+: dotp)
for (i = 0; i < N; i++) {
    dotp += a[i] * b[i];
}
```

複数コアの並列処理の記述例（密行列積）

FUJITSU

- 並列化したいループの直前にOpenMPの指示文を挿入
  - parallel節：スレッドを生成する
  - for節：直後のfor文を対象に並列処理
    privateやsharedで変数の特徴を指定したほうがよい（ループ変数ぐらいであれば不要）

```
for(i=0;i<M;i++){
  for(j=0;j<N;j++){
   for(k=0;k<K;k++){
     C[i][j]+=a[i][k]*b[k][j];
   } } }
```

```
#pragma omp parallel for
for(i=0;i<M;i++){
  for(j=0;j<N;j++){
   for(k=0;k<K;k++){
     C[i][j]+=a[i][k]*b[k][j];
   } } }
```

- M>>Nのときは、コア間の並列処理の効率が高いからこちらがよい

```
for(i=0;i<M;i++){
#pragma omp parallel for
  for(j=0;j<N;j++){
   for(k=0;k<K;k++){
     C[i][j]+=a[i][k]*b[k][j];
   } } }
```

- N>>MのときはNのループを対象にしたほうがよい
- omp parallel節はループ内部に無い方がよいので外に追い出したほうがよい（次ページ）

複数コアの並列処理の記述例（密行列積）

FUJITSU

修正例1:parallel節を外に出す　修正例2: ループの順番の入れ替え

```
for(i=0;i<M;i++){
#pragma omp parallel for
  for(j=0;j<N;j++){
   for(k=0;k<K;k++){
     C[i][j]+=a[i][k]*b[k][j];
   } } }
```

```
#pragma omp parallel
for(i=0;i<M;i++){
#pragma omp for
  for(j=0;j<N;j++){
   for(k=0;k<K;k++){
     C[i][j]+=a[i][k]*b[k][j];
   } } }
```

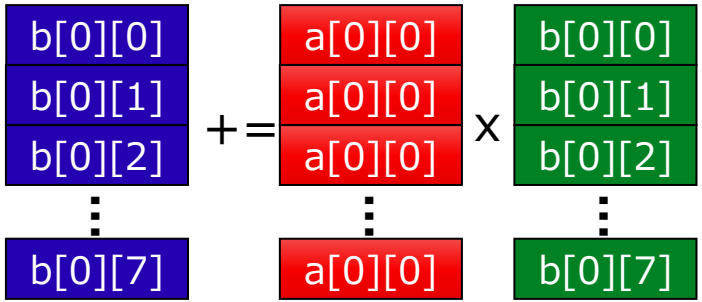
```
#pragma omp parallel for
for(j=0;j<N;j++){
  for(i=0;i<M;i++){
    for(k=0;k<K;k++){
     C[i][j]+=a[i][k]*b[k][j];
  } } }
```

- A64FXではスレッド生成やバリア同期のコストが大きいため、ループ内部にparallel節やfor節はないほうがよい
  - 今回の例では、修正例1のようにparallel節を外に出すだけのではなく、ループの順序を入れ替えるほうがよい
  - ループの順番を入れ替えるのが難しい（かえって性能が落ちる場合）もあるためケースバイケースの判断が必要

（参考）前ページのコードの問題と改善

```
#pragma omp parallel for
for(i=0;i<M;i++){
  for(k=0;k<K;k++){
    for(j=0;j<N;j++){
      c[i][j]+=a[i][k]*b[k][j];
  }}}
```

- aを8要素複製する形でレジスタ上にロードしたい
- ソースコード上で記述すると、メモリ上でのデータの複製になるので遅くなる



さらに高速な行列積を記述したいときは、配列a, b, cをキャッシュヒットするように適切なサイズでブロック化して、最内ループをある程度ループ展開して記述する

コアの並列処理効率の確認

FUJITSU

- ループの回転数が十分あれば、複数コアの並列処理の効率は高いことが多いため、まずはループの回転数が十分かを確認する
  - printf文をソースコード中に挿入するなどして確認
- キャッシュメモリなどの共有資源での競合で性能が向上しないこともあるため、並列化させるコア数を変更して性能を確認
  - デフォルトの設定のOpenMPのスレッド生成数＝コア数
  - スレッド生成数は、環境変数 OMP_NUM_THREADSや omp_set_num_threads(int num)関数で変更可能

コアの並列処理効率の向上

FUJITSU

- 並列化対象のループの回転数を増やす
  - 多重ループであれば複数のループを対象にする（collapse節を挿入）
  - 対象となるループを変更する
  - データ構造とアルゴリズムレベルで見直す
- 細かい最適化をする
  - 立ち上げるスレッド数を減らす
  - omp parallel節の範囲をなるべく広げて、スレッド生成コストを下げる
  - 自動で挿入されるバリア同期を削る（nowait節を挿入）

```
#pragma omp parallel for collapse (2)
for (i = 0; i < M; i++) {
  for (j = 0; j < N; j++) {
   for (k = 0; k < K; k++) {
     C[i][j] += a[i][k] * b[k][j];
} } }
```

FUJITSU

前処理付き勾配法（PCG法）

PCG法の中身について

前処理付き共役勾配法
Preconditioned Conjugate Gradient Method (PCG)

FUJITSU

- 線形方程式の求解方法のひとつ
  - Ax = bを満たすxを求める
  - 反復法
    - 適当な初期解から開始し、繰り返し計算によって真の解に収束させる
    - 今回取り扱う共役勾配法も反復法の一つ
  - 前処理
    - 収束までに要するステップ数を減らすためのテクニックのひとつ
    - 今回は対角スケーリングを実施

Compute $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
for i = 1, 2, ...
  solve $\mathbf{M}\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$
  $\rho^{(i-1)} = \mathbf{r}^{(i-1)}\mathbf{z}^{(i-1)}$
  if i = 1
    $\mathbf{p}^{(1)} = \mathbf{z}^{(0)}$
  else
    $\beta^{(i-1)} = \rho^{(i-1)}/\rho^{(i-2)}$
    $\mathbf{p}^{(i)} = \mathbf{z}^{(i-1)} + \beta^{(i-1)}\mathbf{p}^{(i-1)}$
  end
  if
  $\mathbf{q}^{(i)} = \mathbf{A}\mathbf{p}^{(i)}$
  $\alpha^{(i)} = \rho^{(i-1)}/\mathbf{p}^{(i)}\mathbf{q}^{(i)}$
  $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha^{(i)}\mathbf{p}^{(i)}$
  $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha^{(i)}\mathbf{q}^{(i)}$
  check convergence $|\mathbf{r}|$
end

対角スケーリング

FUJITSU

- もとの行列の対角成分のみを取り出した行列を前処理行列 $\mathbf{M}$ とする
  - $\mathbf{M}$ が3x3の行列とすると

$$\mathbf{M} = \begin{bmatrix} D_1 & 0 & 0 \\ 0 & D_2 & 0 \\ 0 & 0 & D_3 \end{bmatrix}, \quad \mathbf{M}^{-1} = \begin{bmatrix} 1/D_1 & 0 & 0 \\ 0 & 1/D_2 & 0 \\ 0 & 0 & 1/D_3 \end{bmatrix}$$

  - 「$\mathbf{M}\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$を満たすような$\mathbf{z}^{(i-1)}$を求める」というのは、$\mathbf{z}^{(i-1)} = \mathbf{M}^{-1}\mathbf{r}^{(i-1)}$を計算することと同じ
  - 前処理を行うことで、反復法での収束が改善する場合がある

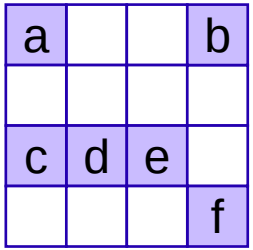
PCG法にて行われる計算

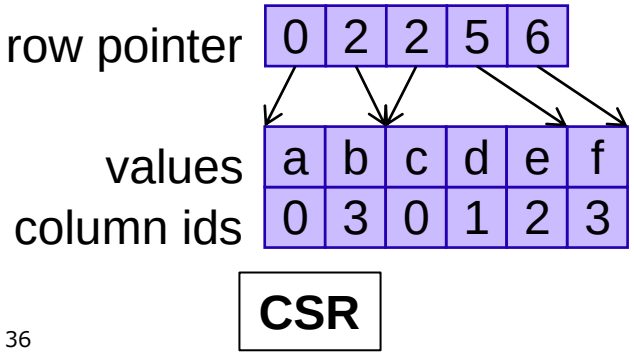
- 行列とベクトルの積計算
  - $\mathbf{Ax}^{(0)}$, $\mathbf{M}^{-1}\mathbf{r}^{(i-1)}$, …
  - 注) 行列は疎な特性を持っている場合有
- ベクトル同士の演算
  - ベクトルの足し算
    - $\mathbf{z}^{(i-1)} + \beta^{(i-1)}\mathbf{p}^{(i-1)}$
    - $\mathbf{x}^{(i-1)} + \alpha^{(i)}\mathbf{p}^{(i)}$
    - …
  - ベクトルの内積
    - $\mathbf{r}^{(i-1)}\mathbf{z}^{(i-1)}$, $\mathbf{p}^{(i)}\mathbf{q}^{(i)}$, …

Compute $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{Ax}^{(0)}$
for i = 1, 2, …
  solve $\mathbf{Mz}^{(i-1)} = \mathbf{r}^{(i-1)}$
  $\rho^{(i-1)} = \mathbf{r}^{(i-1)}\mathbf{z}^{(i-1)}$
  if i = 1
    $\mathbf{p}^{(1)} = \mathbf{z}^{(0)}$
  else
    $\beta^{(i-1)} = \rho^{(i-1)}/\rho^{(i-2)}$
    $\mathbf{p}^{(i)} = \mathbf{z}^{(i-1)} + \beta^{(i-1)}\mathbf{p}^{(i-1)}$
  end
  if
  $\mathbf{q}^{(i)} = \mathbf{Ap}^{(i)}$
  $\alpha^{(i)} = \rho^{(i-1)}/\mathbf{p}^{(i)}\mathbf{q}^{(i)}$
  $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha^{(i)}\mathbf{p}^{(i)}$
  $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha^{(i)}\mathbf{q}^{(i)}$
  check convergence $|\mathbf{r}|$
end

疎行列の格納形式

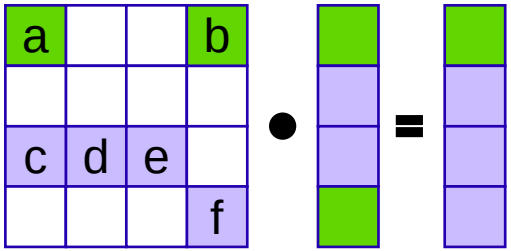
- 疎行列：行列内のほどんど要素の値が0であるような行列
  - 密行列と呼ばれるものの場合、行数をNとした際、要素数はN^2
  - 今回演習で扱うものは、非ゼロ要素数は27 * N程度であり、疎な特性を持つ
- 通常、圧縮したような形で保管をする
  - 計算に寄与しないゼロ要素を保管する必要はない
  - CSR (Compressed Sparse Row) フォーマット
    - 広く用いられている形式であり、行単位で要素を管理する

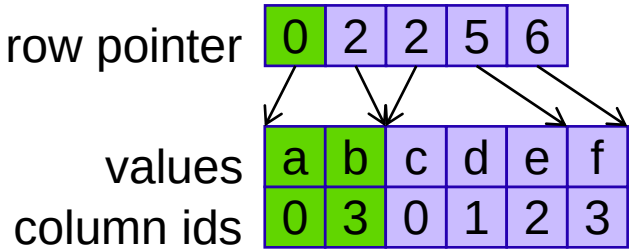




疎行列とベクトルの積計算

- ゼロ以外の行列要素のみを使って計算を行う

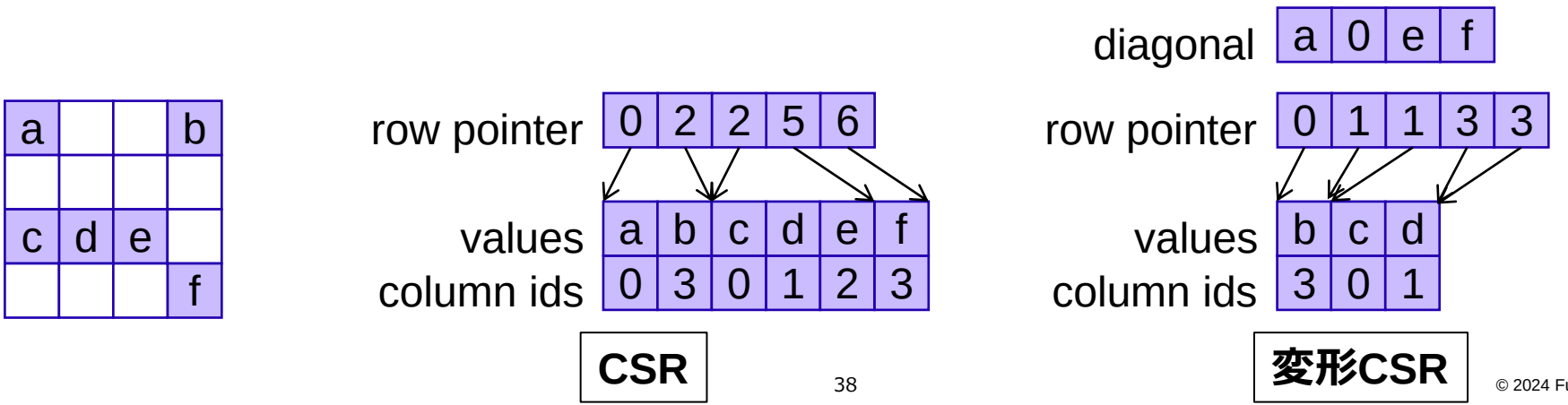





```
for (i = 0; i < N; ++i) {
    double sum = 0.0;
    for (j = rpt[i]; j < rpt[i+1]; ++j) {
        sum += values[j] * x[colids[j]];
    }
    y[i] = sum;
}
```

疎行列の格納形式

- 今回の演習では、対角スケーリングを行う関係上、対角成分と対角成分以外を分けて保管する
  - ここでは、変形CSR形式と呼ぶ
  - 対角成分に関する計算を効率的に行うことが可能



FUJITSU

プログラミング演習

PCG法を例としたOpenMPスレッド並列化

流れ

FUJITSU

- Step 1: ログイン & プログラムの確認
- Step 2: OpenMPによる並列化
- Step 3: 疎行列形式の変更（CSR → ELL） &OpenMPによる並列化
- Step 4: 更なる最適化（スレッド立ち上げの削減）

プログラミング演習

Step 1: ログイン＆プログラムの確認

演習用プログラムの格納先

FUJITSU

- /work/gt00/share/20240205_PCG
  - ジョブスクリプト
    - コンパイル用：compile.sh
    - 実行用：run.sh
  - Makefile
  - ソースコードファイル
    - メイン：poi_csr.c, poi_ell.c
    - 行列生成：gen_poi.c
      - 三次元ポアソン方程式を差分格子のような規則的形状において有限要素法によって離散化して得られる疎行列を係数とする連立一次方程式
    - ソルバー部分：solver_poi.c

演習用プログラムの使い方

- ご自身の領域にコピーしてご使用ください
  - cp -r /work/gt00/share/20240205_PCG /work/gt00/{アカウント名}/.

演習 実行方法

- 作業ディレクトリ
  - /work/gt00/{アカウント名}/20240205_PCG
- コンパイル
  - pjsub compile.sh
- 実行
  - pjsub run.sh
- 実行結果
  - run.sh.{ジョブID}.out

8^3の立方格子での出力例

```
Generate Sparse Matrix A in CSR format
Number of Rows or Columns: 512
Number of non-zero elements: 10648
Initialization (CSR, n = 8), 0.000098 [sec]
0000 step: error is 4.801506e-01
0001 step: error is 3.032428e-01
0002 step: error is 2.079767e-01
0003 step: error is 8.469401e-02
0004 step: error is 2.590772e-02
0005 step: error is 4.320515e-03
0006 step: error is 4.122053e-04
0007 step: error is 5.038052e-05
0008 step: error is 7.018853e-06
0009 step: error is 2.745507e-07
0010 step: error is 2.102801e-08
0011 step: error is 5.216719e-10
Performance (CSR, n = 8), 0.000869 [sec]
```

演習 ジョブスクリプトについて

FUJITSU

- compile.sh
  - Makefileを使用
    - make csr
      - CSR形式にて前処理付き共役勾配法を行うバイナリを生成
    - make ell
      - ELL形式にて前処理付き共役勾配法を行うバイナリを生成
      - step3以降で使用
- run.sh
  - 実行するプログラムを記載する
  - 例）./csr 8
    - compile.shによって生成されたcsrというバイナリを用いる
    - 引数は3次元における各次元あたりの格子点数 (n)
      - n=8の場合、全体の格子点数は512
      - 解くべき線形方程式の係数行列の大きさも512 x 512となる

Step 1 プログラムの性能値

FUJITSU

- 逐次版を実行した際の性能値を確認する
  - 複数の問題サイズで評価をしてみる

実行時間 [秒]	128 ^ 3の立方格子	192 ^3の立方格子
逐次	70.359	349.512

FUJITSU

プログラミング演習

Step 2: OpenMPによる並列化

ここでの目標

- OpenMPの指示文を挿入し、プログラムの並列化を行う
  - solve_poi.c
    - PCG法を行う
    - ファイル内のsolve_poi_CSR関数を対象とする

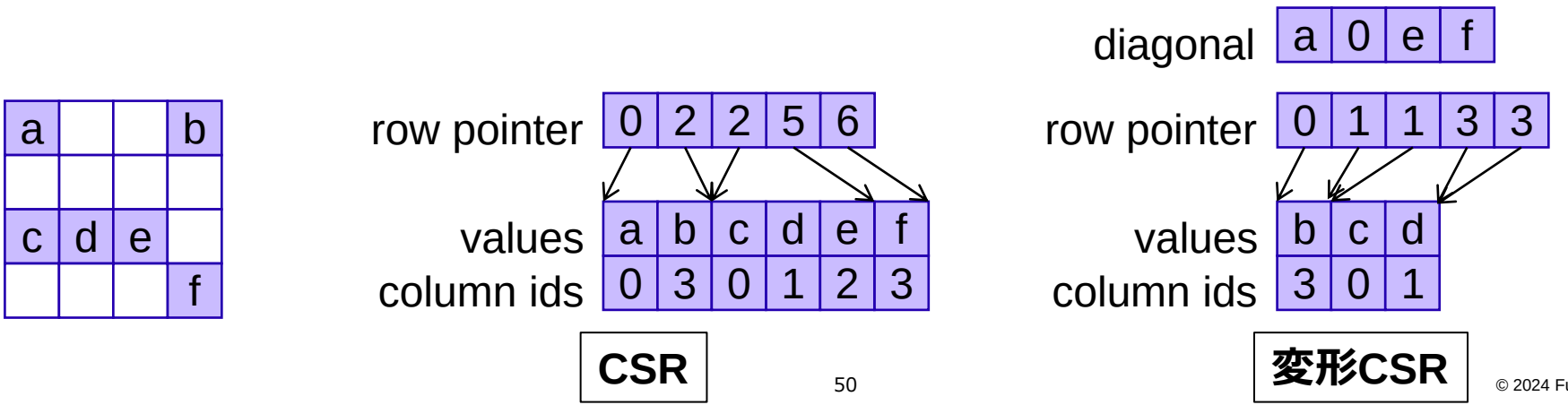
PCG法にて行われる計算（再掲）

- 行列とベクトルの積計算
  - $\mathbf{A}\mathbf{x}^{(0)}, \mathbf{M}^{-1}\mathbf{r}^{(i-1)}, \ldots$
  - 注) 行列は疎な特性を持っている場合有
- ベクトル同士の演算
  - ベクトルの足し算
    - $\mathbf{z}^{(i-1)} + \beta^{(i-1)}\mathbf{p}^{(i-1)}$
    - $\mathbf{x}^{(i-1)} + \alpha^{(i)}\mathbf{p}^{(i)}$
    - …
  - ベクトルの内積
    - $\mathbf{r}^{(i-1)}\mathbf{z}^{(i-1)}$, $\mathbf{p}^{(i)}\mathbf{q}^{(i)}$, …

Compute $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
for i = 1, 2, …
  solve $\mathbf{M}\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$
  $\rho^{(i-1)} = \mathbf{r}^{(i-1)}\mathbf{z}^{(i-1)}$
  if i = 1
    $\mathbf{p}^{(1)} = \mathbf{z}^{(0)}$
  else
    $\beta^{(i-1)} = \rho^{(i-1)}/\rho^{(i-2)}$
    $\mathbf{p}^{(i)} = \mathbf{z}^{(i-1)} + \beta^{(i-1)}\mathbf{p}^{(i-1)}$
  endif
  $\mathbf{q}^{(i)} = \mathbf{A}\mathbf{p}^{(i)}$
  $\alpha^{(i)} = \rho^{(i-1)}/\mathbf{p}^{(i)}\mathbf{q}^{(i)}$
  $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha^{(i)}\mathbf{p}^{(i)}$
  $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha^{(i)}\mathbf{q}^{(i)}$
  check convergence $|\mathbf{r}|$
end

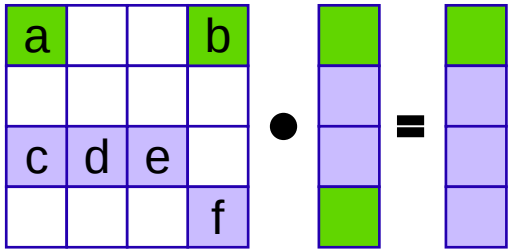
疎行列の格納形式（再掲）

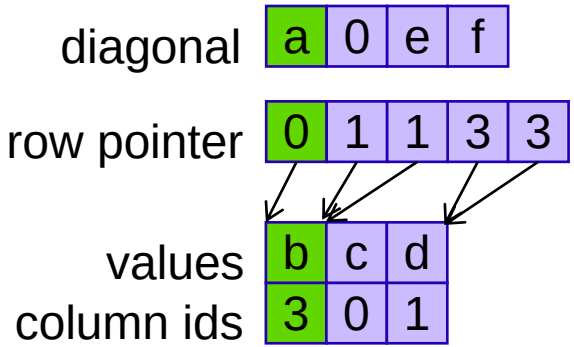
- 今回の演習では、対角スケーリングを行う関係上、対角成分と対角成分以外を分けて保管する
  - ここでは、変形CSR形式と呼ぶ
  - 対角成分に関する計算を効率的に行うことが可能



疎行列とベクトルの積計算

- 変形CSR形式にて計算を行う





変形CSR

```
for (i = 0; i < N; ++i) {
    double sum = dia[i] * x[i];
    for (j = rpt[i]; j < rpt[i+1]; ++j) {
        sum += values[j] * x[colids[j]];
    }
    y[i] = sum;
}
```

ここでの目標

- OpenMPの指示文を挿入し、プログラムの並列化を行う
  - solve_poi.c
    - PCG法を行う
    - ファイル内のsolve_poi_CSR関数を対象とする
      - 内部に含まれている、ベクトルに関する計算や行列に関する計算が対象
      - 補足）
        - privateやreductionを適切に用いる
        - プログラムの修正を加えた際に、結果が変わらないかを確認する
          - 例えば、ステップごとに都度計算しているerrを参照する

Step 2 解答例

FUJITSU

- OpenMP指示文を挿入する

```
    // Compute r = b - Ax
#pragma omp parallel for private(j)
    for (i = 0; i < N; ++i) {
        double sum = A.dia[i] * x[i];
        for (j = A.rpt[i]; j < A.rpt[i + 1]; ++j) {
            sum += A.value[j] * x[A.colid[j]];
        }
        r[i] = b[i] - sum;
    }
    double bnrm2 = 0.0;
#pragma omp parallel for reduction(+:bnrm2)
    for (i = 0; i < N; ++i) {
        bnrm2 += b[i] * b[i];
    }
```

Compute $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
for i = 1, 2, ...
  solve $\mathbf{M}\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$
  $\rho^{(i-1)} = \mathbf{r}^{(i-1)}\mathbf{z}^{(i-1)}$
  if i = 1
    $\mathbf{p}^{(1)} = \mathbf{z}^{(0)}$
  else
    $\beta^{(i-1)} = \rho^{(i-1)}/\rho^{(i-2)}$
    $\mathbf{p}^{(i)} = \mathbf{z}^{(i-1)} + \beta^{(i-1)}\mathbf{p}^{(i-1)}$
  endif
  $\mathbf{q}^{(i)} = \mathbf{A}\mathbf{p}^{(i)}$
  $\alpha^{(i)} = \rho^{(i-1)}/\mathbf{p}^{(i)}\mathbf{q}^{(i)}$
  $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha^{(i)}\mathbf{p}^{(i)}$
  $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha^{(i)}\mathbf{q}^{(i)}$
  check convergence $|\mathbf{r}|$
end

Step 2 解答例

FUJITSU

- OpenMP指示文を挿入する

```
        // Preconditioner
#pragma omp parallel for
        for (i = 0; i < N; ++i) {
            z[i] = r[i] / A.dia[i];
        }
        // rho = r * z
        rho = 0.0;
#pragma omp parallel for reduction(+: rho)
        for (i = 0; i < N; ++i) {
            rho += r[i] * z[i];
        }
```

Compute $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
for i = 1, 2, ...
  solve $\mathbf{M}\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$
  $\rho^{(i-1)} = \mathbf{r}^{(i-1)}\mathbf{z}^{(i-1)}$
  if i = 1
    $\mathbf{p}^{(1)} = \mathbf{z}^{(0)}$
  else
    $\beta^{(i-1)} = \rho^{(i-1)}/\rho^{(i-2)}$
    $\mathbf{p}^{(i)} = \mathbf{z}^{(i-1)} + \beta^{(i-1)}\mathbf{p}^{(i-1)}$
  endif
  $\mathbf{q}^{(i)} = \mathbf{A}\mathbf{p}^{(i)}$
  $\alpha^{(i)} = \rho^{(i-1)}/\mathbf{p}^{(i)}\mathbf{q}^{(i)}$
  $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha^{(i)}\mathbf{p}^{(i)}$
  $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha^{(i)}\mathbf{q}^{(i)}$
  check convergence $|\mathbf{r}|$
end

Step 2 解答例

FUJITSU

- OpenMP指示文を挿入する

```
        if (itr == 0) {
            // p = z
#pragma omp parallel for
            for (i = 0; i < N; ++i) {
                p[i] = z[i];
            }
        }
        else {
            // calculate beta
            // p = z + beta * p
            double beta = rho / rho_prev;
#pragma omp parallel for
            for (i = 0; i < N; ++i) {
                p[i] = z[i] + beta * p[i];
            }
        }
```

Compute $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
for i = 1, 2, ...
  solve $\mathbf{M}\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$
  $\rho^{(i-1)} = \mathbf{r}^{(i-1)}\mathbf{z}^{(i-1)}$
  if i = 1
    $\mathbf{p}^{(1)} = \mathbf{z}^{(0)}$
  else
    $\beta^{(i-1)} = \rho^{(i-1)}/\rho^{(i-2)}$
    $\mathbf{p}^{(i)} = \mathbf{z}^{(i-1)} + \beta^{(i-1)}\mathbf{p}^{(i-1)}$
  endif
  $\mathbf{q}^{(i)} = \mathbf{A}\mathbf{p}^{(i)}$
  $\alpha^{(i)} = \rho^{(i-1)}/\mathbf{p}^{(i)}\mathbf{q}^{(i)}$
  $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha^{(i)}\mathbf{p}^{(i)}$
  $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha^{(i)}\mathbf{q}^{(i)}$
  check convergence $|\mathbf{r}|$
end

Step 2 解答例

FUJITSU

- OpenMP指示文を挿入する

```
#pragma omp parallel for private(j)
      for (i = 0; i < N; ++i) {
         double sum = A.dia[i] * p[i];
         for (j = A.rpt[i]; j < A.rpt[i + 1]; ++j) {
            sum += A.value[j] * p[A.colid[j]];
         }
         q[i] = sum;
      }
```

Compute $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
for i = 1, 2, ...
  solve $\mathbf{M}\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$
  $\rho^{(i-1)} = \mathbf{r}^{(i-1)}\mathbf{z}^{(i-1)}$
  if i = 1
    $\mathbf{p}^{(1)} = \mathbf{z}^{(0)}$
  else
    $\beta^{(i-1)} = \rho^{(i-1)}/\rho^{(i-2)}$
    $\mathbf{p}^{(i)} = \mathbf{z}^{(i-1)} + \beta^{(i-1)}\mathbf{p}^{(i-1)}$
  endif
  $\mathbf{q}^{(i)} = \mathbf{A}\mathbf{p}^{(i)}$
  $\alpha^{(i)} = \rho^{(i-1)}/\mathbf{p}^{(i)}\mathbf{q}^{(i)}$
  $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha^{(i)}\mathbf{p}^{(i)}$
  $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha^{(i)}\mathbf{q}^{(i)}$
  check convergence $|\mathbf{r}|$
end

Step 2 解答例

FUJITSU

- OpenMP指示文を挿入する

```
        double alpha = 0.0;
#pragma omp parallel for reduction(+: alpha)
        for (i = 0; i < N; ++i) {
            alpha += p[i] * q[i];
        }
        alpha = rho / alpha;
        // x = x + alpha * p
#pragma omp parallel for
        for (i = 0; i < N; ++i) {
            x[i] += alpha * p[i];
        }
        // r = r - alpha * q
#pragma omp parallel for
        for (i = 0; i < N; ++i) {
            r[i] -= alpha * q[i];
        }
```

Compute $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
for i = 1, 2, ...
  solve $\mathbf{M}\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$
  $\rho^{(i-1)} = \mathbf{r}^{(i-1)}\mathbf{z}^{(i-1)}$
  if i = 1
    $\mathbf{p}^{(1)} = \mathbf{z}^{(0)}$
  else
    $\beta^{(i-1)} = \rho^{(i-1)}/\rho^{(i-2)}$
    $\mathbf{p}^{(i)} = \mathbf{z}^{(i-1)} + \beta^{(i-1)}\mathbf{p}^{(i-1)}$
  endif
  $\mathbf{q}^{(i)} = \mathbf{A}\mathbf{p}^{(i)}$
  $\alpha^{(i)} = \rho^{(i-1)}/\mathbf{p}^{(i)}\mathbf{q}^{(i)}$
  $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha^{(i)}\mathbf{p}^{(i)}$
  $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha^{(i)}\mathbf{q}^{(i)}$
  check convergence $|\mathbf{r}|$
end

Step 2 解答例

FUJITSU

- OpenMP指示文を挿入する

```
        double dnrm2 = 0.0;
#pragma omp parallel for reduction(+: dnrm2)
        for (i = 0; i < N; ++i) {
            dnrm2 += r[i] * r[i];
        }
        double err = sqrt(dnrm2 / bnrm2);
```

Compute $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
for i = 1, 2, ...
    solve $\mathbf{M}\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$
    $\rho^{(i-1)} = \mathbf{r}^{(i-1)}\mathbf{z}^{(i-1)}$
    if i = 1
        $\mathbf{p}^{(1)} = \mathbf{z}^{(0)}$
    else
        $\beta^{(i-1)} = \rho^{(i-1)}/\rho^{(i-2)}$
        $\mathbf{p}^{(i)} = \mathbf{z}^{(i-1)} + \beta^{(i-1)}\mathbf{p}^{(i-1)}$
    endif
    $\mathbf{q}^{(i)} = \mathbf{A}\mathbf{p}^{(i)}$
    $\alpha^{(i)} = \rho^{(i-1)}/\mathbf{p}^{(i)}\mathbf{q}^{(i)}$
    $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha^{(i)}\mathbf{p}^{(i)}$
    $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha^{(i)}\mathbf{q}^{(i)}$
    check convergence $|\mathbf{r}|$
end

Step 2 プログラムの性能値

FUJITSU

- OpenMPによって並列化したプログラムを実行した際の性能値を確認する
  - 12スレッド (1CMG) のみを使用した場合の性能と全コア (48コア, 4CMG) を使用した場合の性能の両方を確認すると良い

実行時間 [秒]		128 ^ 3の立方格子	192 ^ 3の立方格子
逐次		70.359	349.512
OpenMP並列化	12スレッド実行	6.567	32.031
	48スレッド実行	4.681	16.410

プログラミング演習

Step 3: 疎行列形式の変更（CSR → ELL）＆OpenMPによる並列化

ここでの目標

- 疎行列の格納形式を変えた際にどの程度性能が変わるのかを見る
- さらに、OpenMPの指示文を挿入し、プログラムの並列化を行う
  - poi_ell.c
    - 全体とりまとめをしている。main関数を含む
      - CSRを使用する際には、poi_csr.cが使用される
  - gen_poi.c
    - 行列やベクトルの初期化等を行っている
      - 3次元ポアソン方程式の離散化
    - ファイル内のinitialize_ELL関数を対象とする
  - solve_poi.c
    - PCG法を行う
    - ファイル内のsolve_poi_ELL関数を対象とする

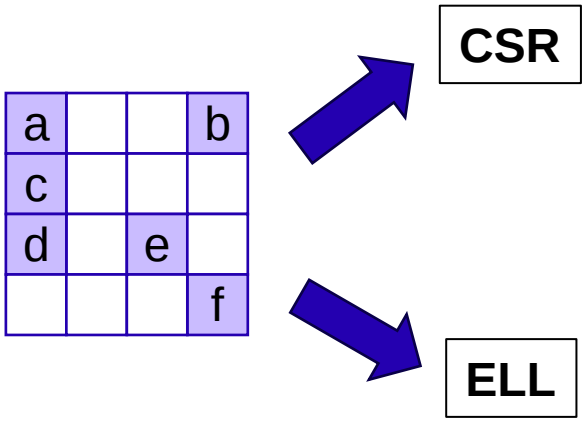
プログラム内でのELLフォーマット使用方法

FUJITSU

- compile.shを修正
  - make csr の箇所を make ell とする
  - → ELL形式を使用した実行ファイルが生成される

- run.shを修正
  - ./csr 128 の箇所を ./ell 128 とする
  - → ELL形式を使用した実行ファイルが実行される

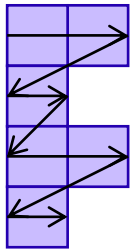
ELLフォーマット



row pointer: 0 2 2 5 6

values: a b c d e f

column ids: 0 3 0 1 2 3



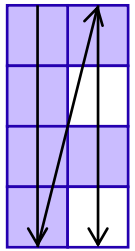
行方向に連続
(Row-major)

イメージとしては
1. 各行にある非ゼロ要素を左詰め
2. 最大幅に合わせて0-padding
3. 列方向に連続となるようにメモリ配置

width: 2

values: a c d f b e

column ids: 0 0 0 3 3 2



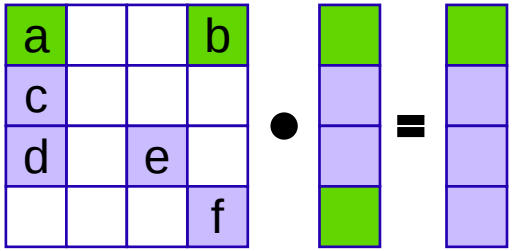
列方向に連続
(Column-major)

ゼロに関する無駄な計算が増える一方で、SIMD効率Up

ELLフォーマットでの行列とベクトルの積計算

FUJITSU

- 保管されている情報のみから計算を行う



width 2

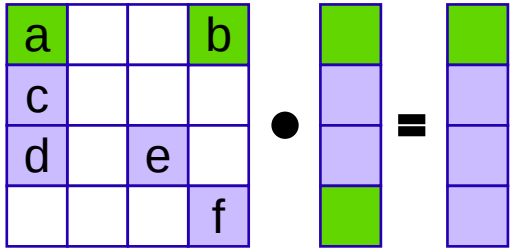
values	a	c	d	f	b		e	
column ids	0	0	0	3	3		2	

```
for (i = 0; i < N; ++i) {
    y[i] = 0;
}
for (j = 0; j < width; ++j) {
    for (i = 0; i < N; ++i) {
      y[i] += values[i + j * N] * x[colids[i + j * N]];
    }
}
```

Nが十分に大きい場合に、SIMDの効率やスレッド並列化の効果を大きくできるのがELLの利点

変形ELLの導入

- 今回の演習では、対角スケーリングを行う関係上、対角成分と対角成分以外を分けて保管する
  - ここでは、変形ELL形式と呼ぶ



diagonal	a		e	f
width	1			
values	b	c	d	
column ids	3	0	0	

```
for (i = 0; i < N; ++i) {
    y[i] = dia[i] * x[i];
}
for (j = 0; j < width; ++j) {
    for (i = 0; i < N; ++i) {
        y[i] += values[i + j * N] * x[colids[i + j * N]];
    }
}
```

ここでの目標（再掲）

- 疎行列の格納形式を変えた際にどの程度性能が変わるのかを見る。さらに、OpenMPの指示文を挿入し、プログラムの並列化を行う
  - poi_ell.c
    - 全体とりまとめをしている。main関数を含む
      - CSRを使用する際には、poi_csr.cが使用される
  - gen_poi.c
    - 行列やベクトルの初期化等を行っている
      - 3次元ポアソン方程式の離散化
    - **ファイル内のinitialize_ELL関数を対象とする**
  - solve_poi.c
    - PCG法を行う
    - **ファイル内のsolve_poi_ELL関数を対象とする**

Step 3 解答例 (gen_poi.c)

FUJITSU

- OpenMP指示文を挿入する

```
 // initialize colid and value
#pragma omp parallel for private (j)
    for (i = 0; i < A->width; ++i) {
        for (j = 0; j < A->nrow; ++j) {
            A->colid[j + i * A->nrow] = j;
            A->value[j + i * A->nrow] = 0.0;
        }
    }
```

```
 // initialize colid and value
#pragma omp parallel for collapse(2)
    for (i = 0; i < A->width; ++i) {
        for (j = 0; j < A->nrow; ++j) {
         ...}}
```

Step 3 解答例 (gen_poi.c)

FUJITSU

- OpenMP指示文を挿入する

```
int cur_nnz = 0;
#pragma omp parallel for private(j, k) reduction(+:cur_nnz)
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            for (k = 0; k < n; ++k) {
                int cur_row = i * n * n + j * n + k;
                ...
                cur_nnz += cur;
                x[cur_row] = 0.0;
                b[cur_row] = 27.0 - cur - 1;
            }
        }
    }
```

Step 3 解答例 (gen_poi.c)

FUJITSU

- collapse文を使用することも可能

```
int cur_nnz = 0;
#pragma omp parallel for collapse(3) reduction(+:cur_nnz)
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            for (k = 0; k < n; ++k) {
                int cur_row = i * n * n + j * n + k;
                ...
                cur_nnz += cur;
                x[cur_row] = 0.0;
                b[cur_row] = 27.0 - cur - 1;
            }
        }
    }
```

ijkループによって得られる
cur_rowは互いに異なるため、
並列化が可能
→ ループ長をより長くするために、
collapse文を使用する

Step 3 プログラムの性能値 (gen_poi.c)

FUJITSU

- 行列生成部分の性能を逐次とOpenMP並列化の両方で確認する

実行時間 [秒]		128 ^3の立方格子	192 ^3の立方格子
逐次		7.157	11.152
OpenMP並列化（collapse導入）	12スレッド	0.700	0.768
	48スレッド	0.329	0.234

Step 3 解答例 (solve_poi.c)

FUJITSU

- OpenMP指示文を挿入する

```
   // Compute r = b - Ax
#pragma omp parallel for
   for (i = 0; i < N; ++i) {
      r[i] = A.dia[i] * x[i];
   }
   for (j = 0; j < A.width; ++j) {
#pragma omp parallel for
      for (i = 0; i < N; ++i) {
         r[i] += A.value[i +  j * N] * x[A.colid[i + j * N]];
      }
   }
#pragma omp parallel for
   for (i = 0; i < N; ++i) {
      r[i] = b[i] - r[i];
   }
```

Compute $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
for i = 1, 2, ...
  solve $\mathbf{M}\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$
  $\rho^{(i-1)} = \mathbf{r}^{(i-1)}\mathbf{z}^{(i-1)}$
  if i = 1
    $\mathbf{p}^{(1)} = \mathbf{z}^{(0)}$
  else
    $\beta^{(i-1)} = \rho^{(i-1)}/\rho^{(i-2)}$
    $\mathbf{p}^{(i)} = \mathbf{z}^{(i-1)} + \beta^{(i-1)}\mathbf{p}^{(i-1)}$
  endif
  $\mathbf{q}^{(i)} = \mathbf{A}\mathbf{p}^{(i)}$
  $\alpha^{(i)} = \rho^{(i-1)}/\mathbf{p}^{(i)}\mathbf{q}^{(i)}$
  $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha^{(i)}\mathbf{p}^{(i)}$
  $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha^{(i)}\mathbf{q}^{(i)}$
  check convergence $|\mathbf{r}|$
end

Step 3 解答例 (solve_poi.c)

FUJITSU

- OpenMP指示文を挿入する

```
      // q = Ap
#pragma omp parallel for
      for (i = 0; i < N; ++i) {
          q[i] = A.dia[i] * p[i];
      }
      for (j = 0; j < A.width; ++j) {
#pragma omp parallel for
          for (i = 0; i < N; ++i) {
              q[i] += A.value[i +  j * N] * p[A.colid[i + j *
N]];
          }
      }
```

Compute $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
for i = 1, 2, ...
  solve $\mathbf{M}\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$
  $\rho^{(i-1)} = \mathbf{r}^{(i-1)}\mathbf{z}^{(i-1)}$
  if i = 1
    $\mathbf{p}^{(1)} = \mathbf{z}^{(0)}$
  else
    $\beta^{(i-1)} = \rho^{(i-1)}/\rho^{(i-2)}$
    $\mathbf{p}^{(i)} = \mathbf{z}^{(i-1)} + \beta^{(i-1)}\mathbf{p}^{(i-1)}$
  endif
  $\mathbf{q}^{(i)} = \mathbf{A}\mathbf{p}^{(i)}$
  $\alpha^{(i)} = \rho^{(i-1)}/\mathbf{p}^{(i)}\mathbf{q}^{(i)}$
  $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha^{(i)}\mathbf{p}^{(i)}$
  $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha^{(i)}\mathbf{q}^{(i)}$
  check convergence $|\mathbf{r}|$
end

Step 3 プログラムの性能値 (solve_poi.c)

FUJITSU

- ELLを使用した場合の性能を逐次とOpenMP並列化の両方で確認する

実行時間 [秒]		128 ^ 3の立方格子	192 ^3の立方格子
逐次 (CSR)		70.359	349.512
OpenMP並列化 (CSR)	12スレッド	6.567	32.031
	48スレッド	4.681	16.410
逐次 (ELL)		31.638	153.040
OpenMP並列化 (ELL)	12スレッド	3.923	19.735
	48スレッド	1.757	6.568

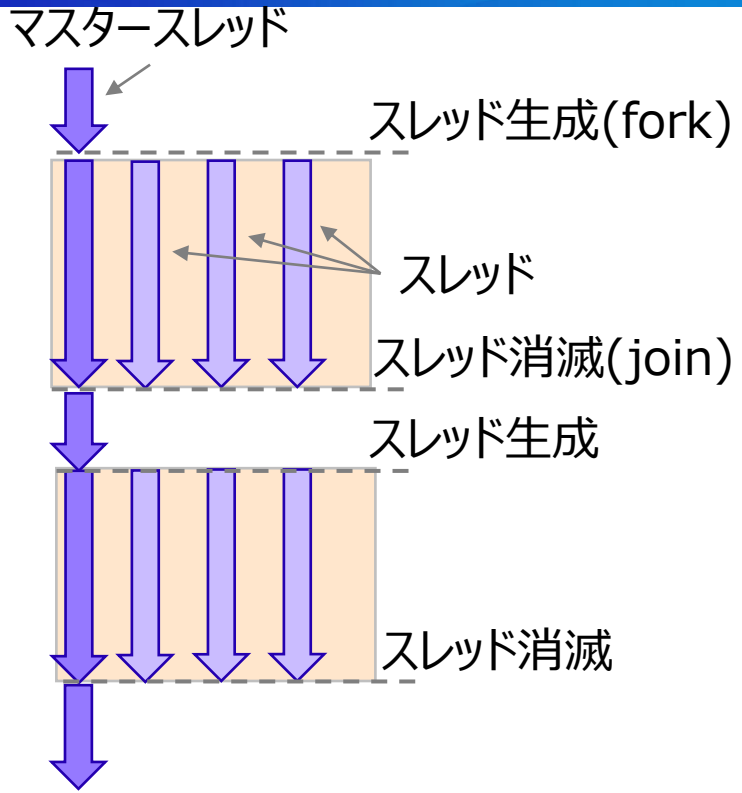
FUJITSU

プログラミング演習

Step 4: 更なる最適化（スレッド立ち上げの削減）

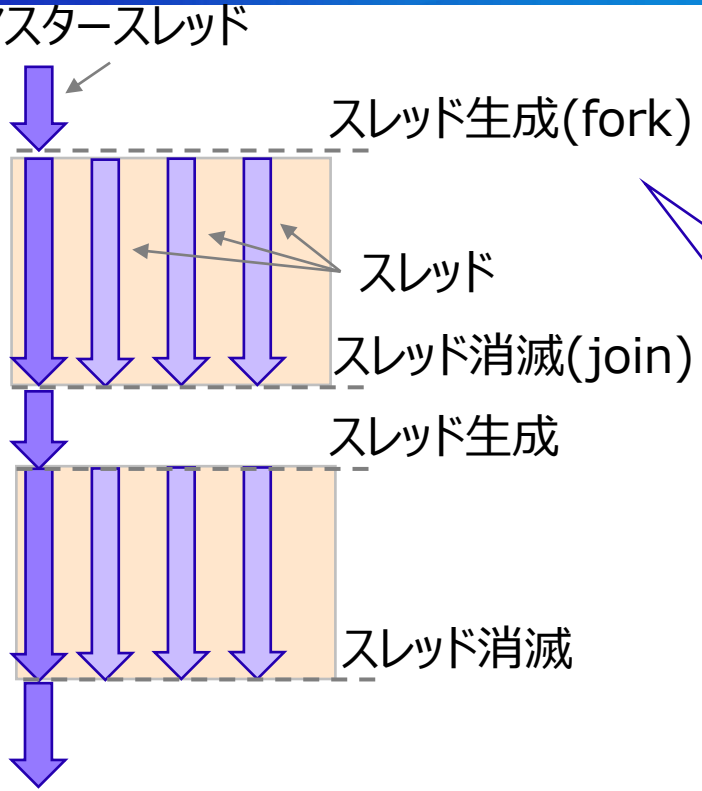
OpenMPでよく記述される並列モデル（再掲）

```
// 逐次処理
…
#pragma omp parallel
{
   // 並列処理A
   …
}
// 逐次処理
…
#pragma omp parallel
{
  // 並列処理B
  …
}
// 逐次処理
…
```



OpenMPでよく記述される並列モデル

```
// 逐次処理
…
#pragma omp parallel
{
   // 並列処理A
   …
}
// 逐次処理
…
#pragma omp parallel
{
  // 並列処理B
  …
}
// 逐次処理
…
```



omp parallelが呼ばれるたびにスレッドの生成と消滅が行われる
→
頻繁にomp parallelが呼ばれると、スレッド生成/消滅がオーバーヘッドになる

omp parallel 削減例

```
#pragma omp parallel for
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      a[i] = b[i] + c[i];
    }
#pragma omp parallel for
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      d[i] = b[i] + c[i];
    }

#pragma omp parallel for reduction
(+:sum)
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      sum += b[i] + c[i];
    }
```

63 msec

```
#pragma omp parallel
    {

#pragma omp for
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      a[i] = b[i] + c[i];
    }
#pragma omp for
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      d[i] = b[i] + c[i];
    }
#pragma omp for reduction (+:sum)
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      sum += b[i] + c[i];
    }
    }
```

38 msec

スレッド生成回数が少ない右のコードのほうが高速

更なるオーバーヘッド削減のために

```
#pragma omp parallel
    {

#pragma omp for nowait
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      a[i] = b[i] + c[i];
    }
#pragma omp for nowait
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      d[i] = b[i] + c[i];
    }
#pragma omp for reduction (+:sum) nowait
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      sum += b[i] + c[i];
    }

    }
```

- nowait節をつけることで暗黙のバリア同期を削れる
- 38msec → 25msecへと改善

ここでの目標

- プログラムの更なる最適化を行う
  - ELLフォーマットを対象とする
  - メインとしてはソルバーを取り扱う
    - solve_poi.c
      - PCG法を行う
      - ファイル内のsolve_poi_ELL関数を対象とする

Step 4 解答例 (solve_poi.c)

FUJITSU

- OpenMP指示文を挿入する

```
 // Compute r = b - Ax
#pragma omp parallel private(i, j)
   {
#pragma omp for nowait
       for (i = 0; i < N; ++i) {
           r[i] = A.dia[i] * x[i];
       }
       for (j = 0; j < A.width; ++j) {
#pragma omp for nowait
           for (i = 0; i < N; ++i) {
               r[i] += A.value[i +  j * N] * x[A.colid[i + j * N]];
           }
       }
#pragma omp for nowait
       for (i = 0; i < N; ++i) {
           r[i] = b[i] - r[i];
       }
   }
```

Compute $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
for i = 1, 2, ...
    solve $\mathbf{M}\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$
    $\rho^{(i-1)} = \mathbf{r}^{(i-1)}\mathbf{z}^{(i-1)}$
    if i = 1
        $\mathbf{p}^{(1)} = \mathbf{z}^{(0)}$
    else
        $\beta^{(i-1)} = \rho^{(i-1)}/\rho^{(i-2)}$
        $\mathbf{p}^{(i)} = \mathbf{z}^{(i-1)} + \beta^{(i-1)}\mathbf{p}^{(i-1)}$
    endif
    $\mathbf{q}^{(i)} = \mathbf{A}\mathbf{p}^{(i)}$
    $\alpha^{(i)} = \rho^{(i-1)}/\mathbf{p}^{(i)}\mathbf{q}^{(i)}$
    $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha^{(i)}\mathbf{p}^{(i)}$
    $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha^{(i)}\mathbf{q}^{(i)}$
    check convergence $|\mathbf{r}|$
end

Step 4 解答例 (solve_poi.c)

FUJITSU

- OpenMP指示文を挿入する

```
      // q = Ap
#pragma omp parallel private(i, j)
      {
#pragma omp for nowait
          for (i = 0; i < N; ++i) {
              q[i] = A.dia[i] * p[i];
          }
          for (j = 0; j < A.width; ++j) {
#pragma omp for nowait
              for (i = 0; i < N; ++i) {
                  q[i] += A.value[i +  j * N] * p[A.colid[i + j * N]];
              }
          }
      }
```

Compute $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
for i = 1, 2, ...
  solve $\mathbf{M}\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$
  $\rho^{(i-1)} = \mathbf{r}^{(i-1)}\mathbf{z}^{(i-1)}$
  if i = 1
    $\mathbf{p}^{(1)} = \mathbf{z}^{(0)}$
  else
    $\beta^{(i-1)} = \rho^{(i-1)}/\rho^{(i-2)}$
    $\mathbf{p}^{(i)} = \mathbf{z}^{(i-1)} + \beta^{(i-1)}\mathbf{p}^{(i-1)}$
  endif
  $\mathbf{q}^{(i)} = \mathbf{A}\mathbf{p}^{(i)}$
  $\alpha^{(i)} = \rho^{(i-1)}/\mathbf{p}^{(i)}\mathbf{q}^{(i)}$
  $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha^{(i)}\mathbf{p}^{(i)}$
  $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha^{(i)}\mathbf{q}^{(i)}$
  check convergence $|\mathbf{r}|$
end

Step 4 プログラムの性能値 (solve_poi.c)

FUJITSU

- 最適化を追加した場合の性能を確認する

実行時間 [秒]		128 ^ 3の立方格子	192 ^ 3の立方格子
逐次 (CSR)		70.359	349.512
OpenMP並列化 (CSR)	12スレッド	6.567	32.031
	48スレッド	4.681	16.410
逐次 (ELL)		31.638	153.040
OpenMP並列化 (ELL)	12スレッド	3.923	19.735
	48スレッド	1.757	6.568
OpenMP並列化 (ELL, +最適化)	12スレッド	3.748	17.663
	48スレッド	1.243	6.083

まとめ

- 離散化した3次元ポアソン方程式を例に、連立一次方程式の求解を実現する前処理付き共役勾配法を紹介
  - Wisteria/BDEC-1 (Odyssey) での実習
    - 多数の計算コアと高いメモリバンド幅を有するA64FXを使用
    - OpenMP並列化によってマルチコアを使用
    - データレイアウトの工夫により、SIMD化率の向上と高メモリバンド幅を活用
    - OpenMP並列化で更に性能を出すための方法を紹介

FUJITSU