

# KOBE HPCサマースクール (初級)2026

講義資料 2026/09/14

兵庫県立大学 情報科学研究科  
安田 修悟

# 背景と講義目的

- 背景

「富岳」のような大規模並列スーパーコンピュータによるシミュレーションを活用するためには、並列アプリケーションの開発に必要な基本的な並列計算法についての知識が必要となる。

- 講義目的

並列計算についての知識と基礎的な並列計算のプログラミング技術の習得を目的とする。

- 各種並列計算（スレッド並列、プロセス並列、アクセラレータ）について、それぞれ具体的なプログラミング法を学習する。
- 高速化のためのプログラミング技術についても学習する。

# 講義内容

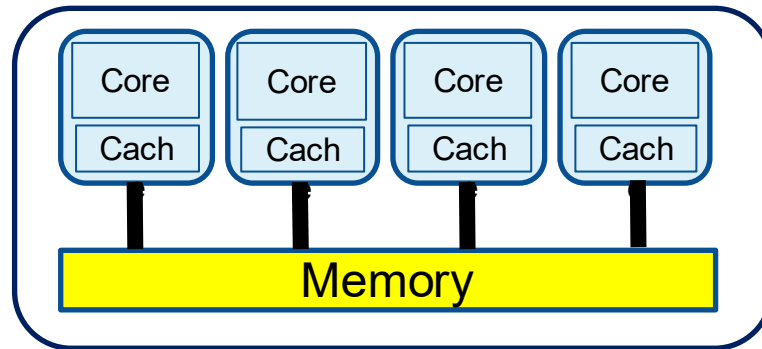
1. 計算機サーバーの環境設定と使い方 (担当 兵県大・安田)
2. シリアルプログラムの高速化
3. 熱伝達問題の差分計算
4. スレッド並列とは (担当 R-CCS・今村先生)
5. OpenMPによるループ処理の並列化
6. (招待講演: Mohamed Wahib先生)
7. 差分化された偏微分方程式の並列化
8. アムダール法則と並列化効率の評価
9. 分散メモリ型並列計算機とは何か? (担当 神大・三宅先生)
10. 1対1通信関数、集団通信関数
11. 並列計算性能の評価方法
12. 熱伝導問題のプロセス並列計算
13. ハイブリッド並列
14. アクセラレータとは (担当 兵県大・安田)
15. OpenACCプログラミング
16. まとめと確認テスト

# 並列計算機と並列計算の種類

# 並列計算機

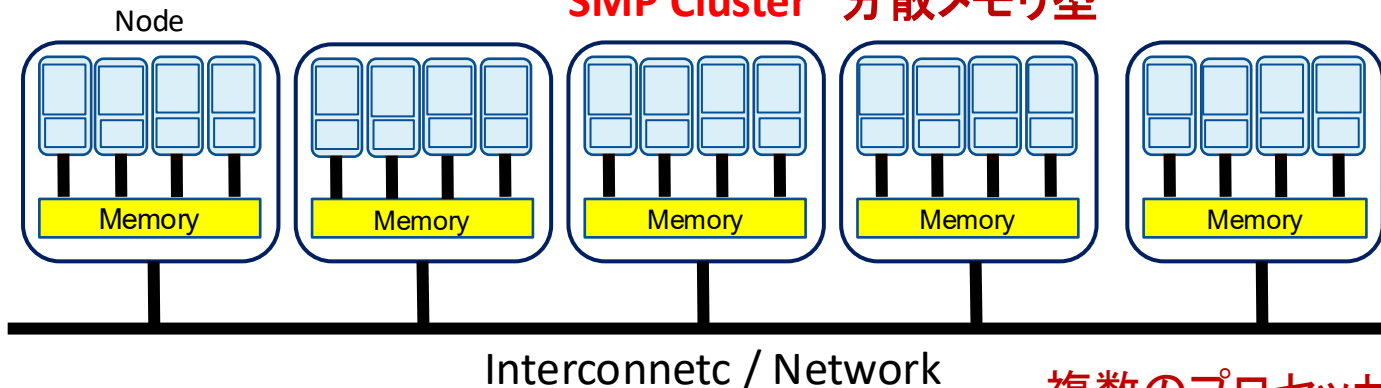
- 計算機アーキテクチャ

## SMP (Symmetric Multiprocessing) 共有メモリ型



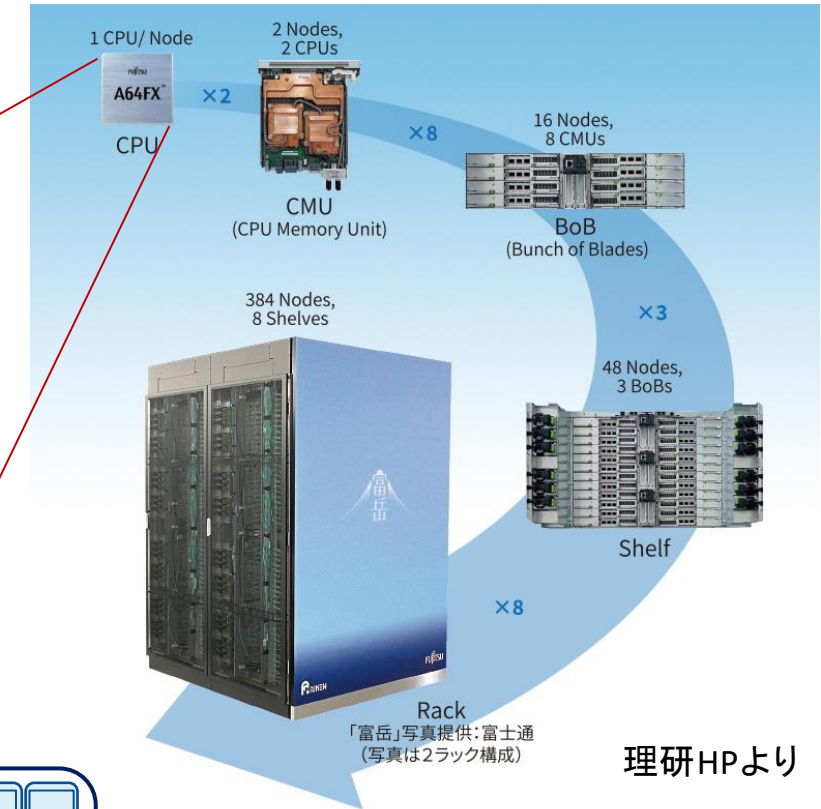
複数のコアが均一に並列処理

## SMP Cluster 分散メモリ型



A64FX CPUダイ

## 富岳の構成

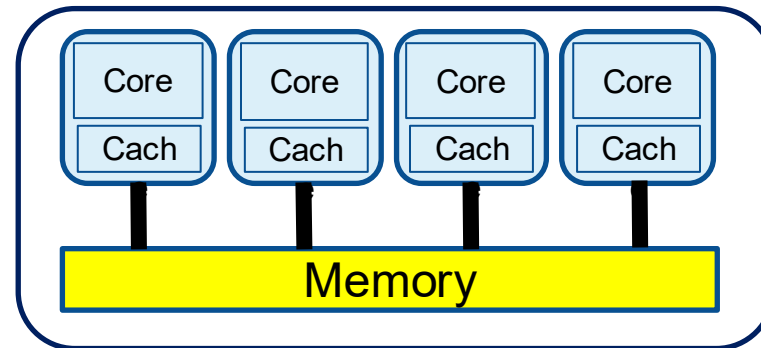


複数のプロセッサがネットワークを介して通信

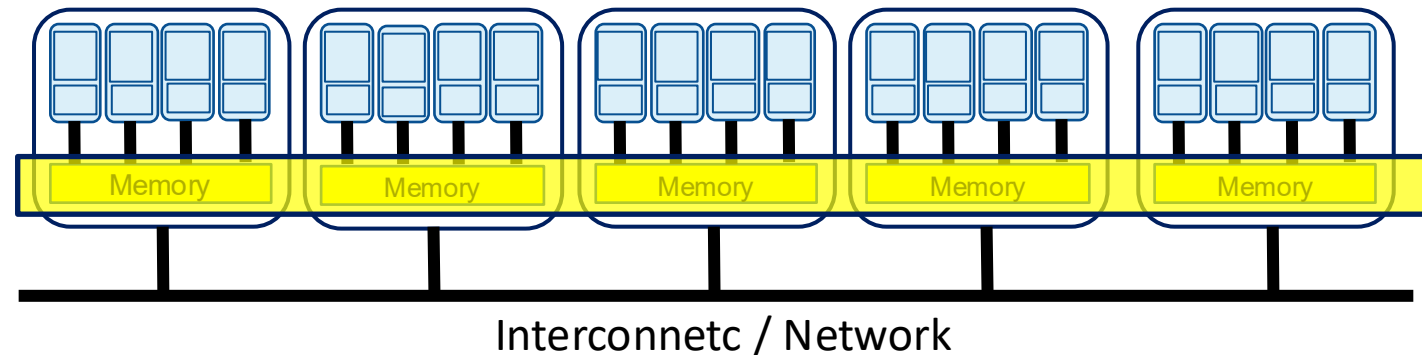
# 並列計算機

- 計算機アーキテクチャ

SMP (Symmetric Multiprocessing) 共有メモリ型



NUMA (Non-Uniform Memory Access) 共有メモリ型マルチプロセッサ計算システム

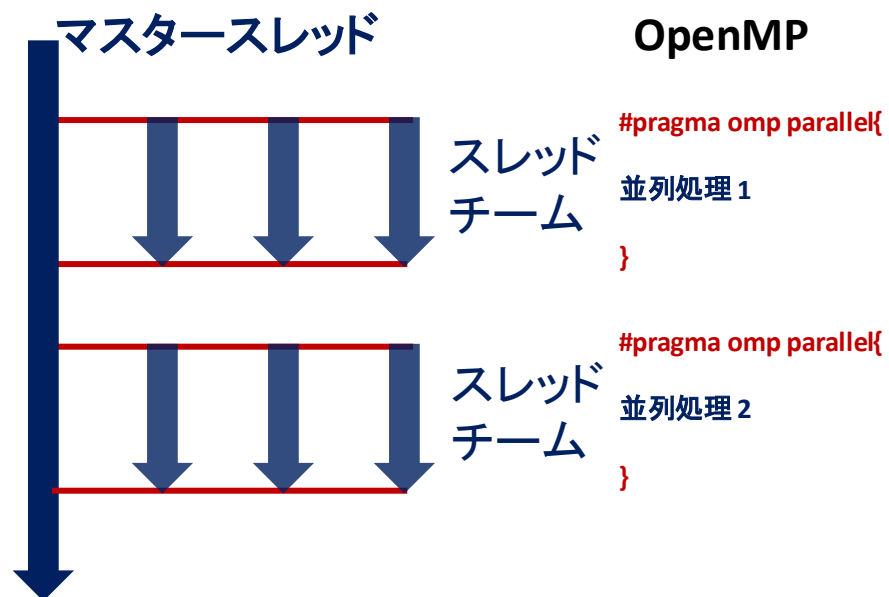


# 並列計算とは

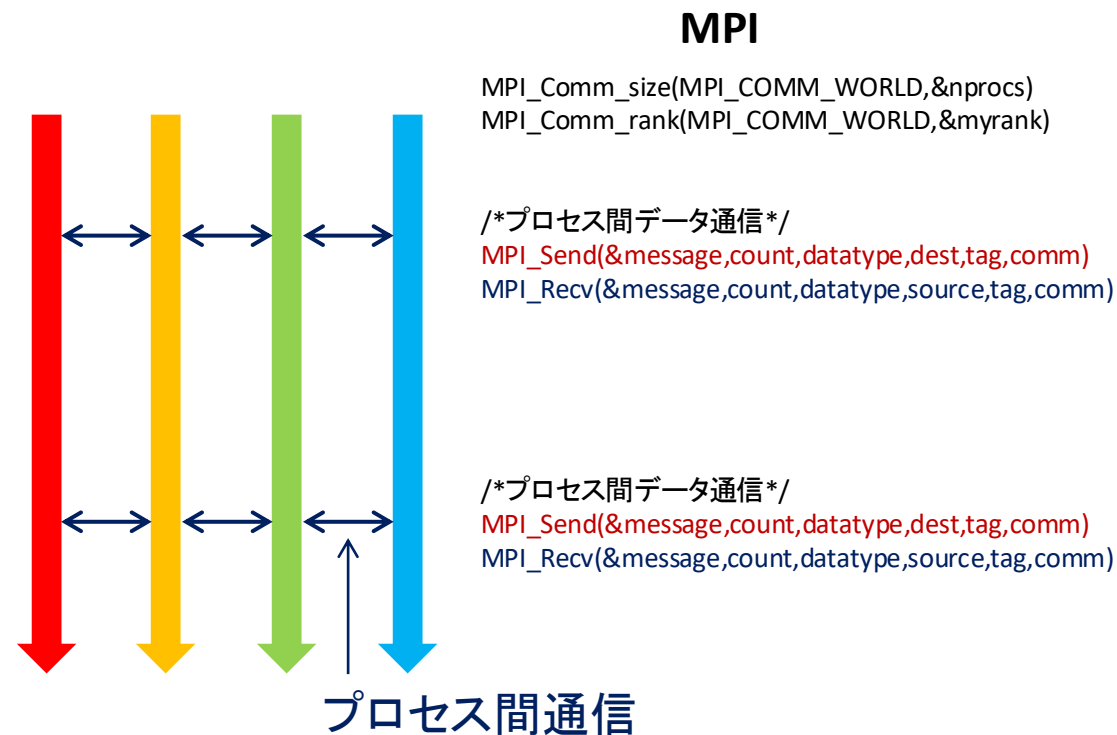
- 並列計算・・・演算を分割して、複数のプロセッサに割り当てて処理する
  - スレッド並列（共有メモリ型）
    - 分割された演算（スレッド）が、メモリ空間を共有できる。
    - 複数のスレッドが均一に並列処理をする。
  - プロセス並列（分散メモリ型）
    - 分割された演算（プロセス）が、それぞれ独立のメモリ空間を参照。
    - プロセス間にデータ転送が必要。

# 並列計算の実行イメージ

- スレッド並列



- プロセス並列



# 並列計算機と並列計算の種類

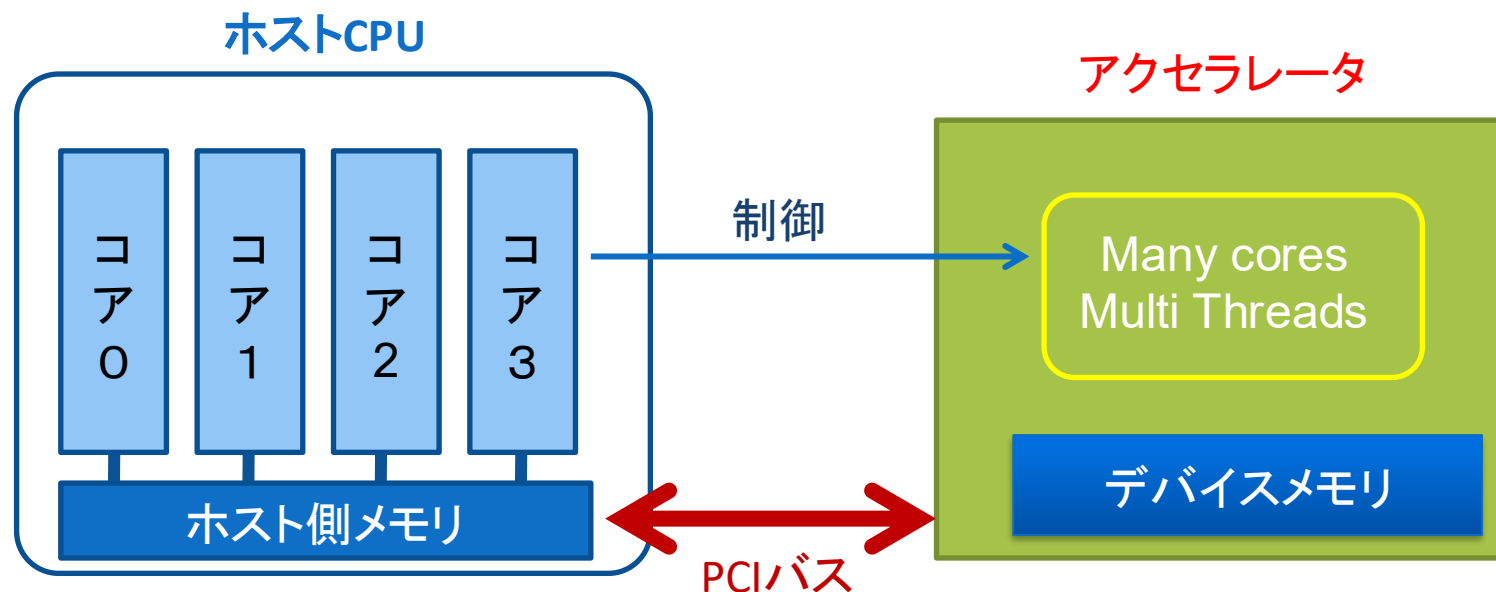
|          | SMP                                                                     | NUMA                                                                                   | Cluster                                                                                                |
|----------|-------------------------------------------------------------------------|----------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| 特徴       | <ul style="list-style-type: none"><li>メモリを共有</li><li>データ転送が不要</li></ul> | <ul style="list-style-type: none"><li>論理的にメモリ共有(物理的には分散)</li><li>明示的データ通信は不要</li></ul> | <ul style="list-style-type: none"><li>ローカルにメモリ</li><li>ノード間データ転送が必要</li></ul>                          |
| プログラミング  | <ul style="list-style-type: none"><li>容易<br/>自動並列化、OpenMPが可能</li></ul>  | <ul style="list-style-type: none"><li>容易<br/>自動並列化、OpenMPが可能(性能にはメモリ配置の影響あり)</li></ul> | <ul style="list-style-type: none"><li>難しい<br/>MPIによるノード間通信が必要。<br/>(自動並列化、OpenMPIはノード内でのみ可能)</li></ul> |
| スケーラビリティ | 中                                                                       | 高い                                                                                     | 非常に高い                                                                                                  |

# アクセラレータ

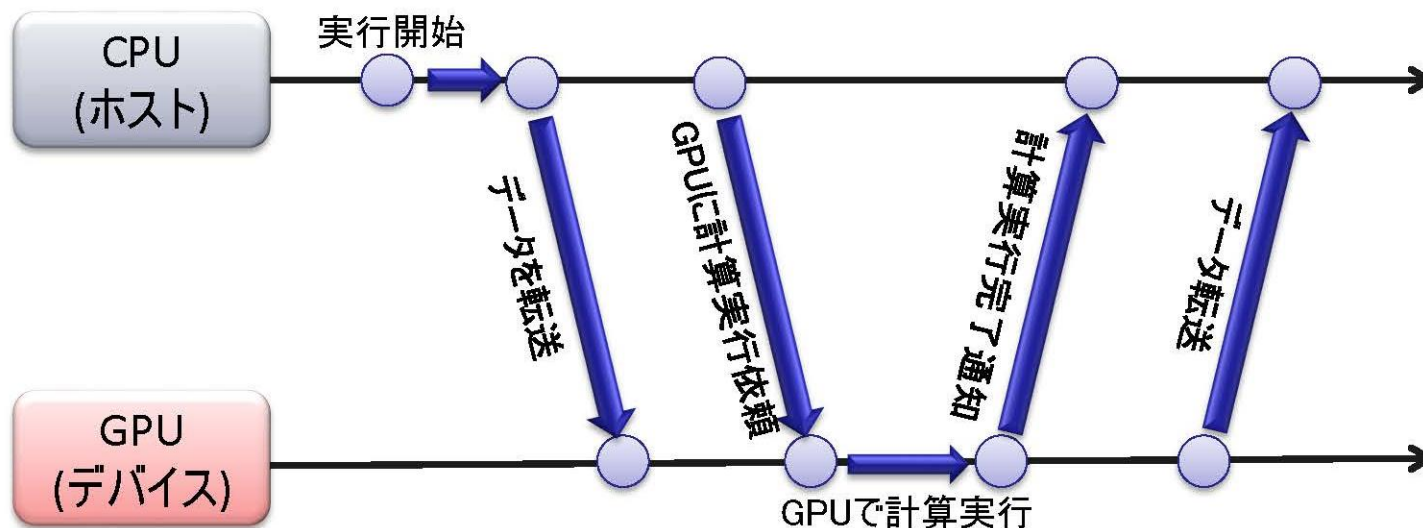
- CPUの処理を一部代替して全体の処理の効率を向上させる装置



## ハイブリット構成（CPU＋アクセラレータ）



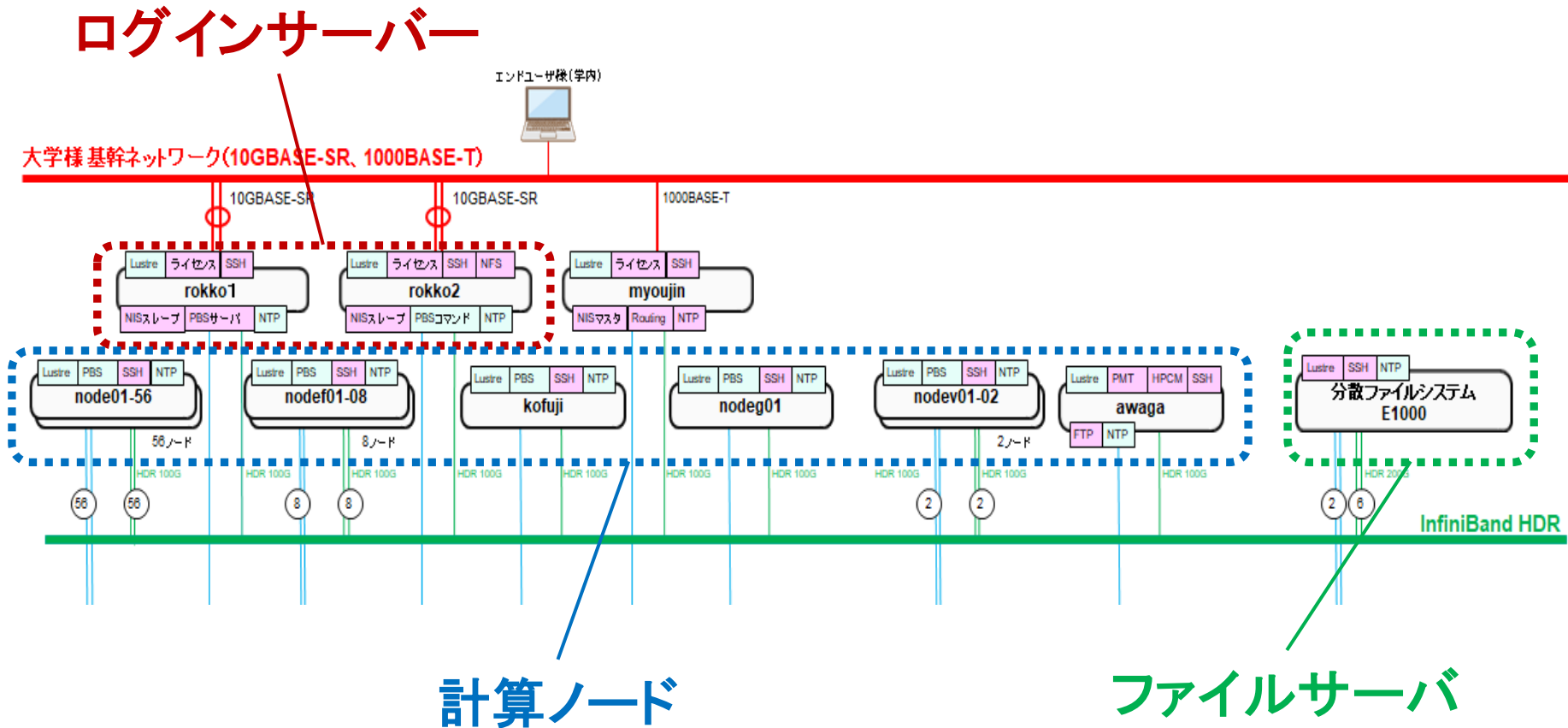
# アクセラレータの実行イメージ



- GPU(デバイス)ではCPU(ホスト)から制御をします。
- GPU(デバイス)の計算に必要なデータはCPU(ホスト)から転送します。
- GPU(デバイス)で計算した結果はCPU(ホスト)に転送します。

# 計算機サーバと環境設定

# 兵庫県大・情報科学キャンパス 計算機サーバシステム概要



# 計算機サーバシステム概要

|                         | Host                    | System                      | CPU/Accelerator                                                                             | Memory                       |
|-------------------------|-------------------------|-----------------------------|---------------------------------------------------------------------------------------------|------------------------------|
| フロントエンドサーバ              | rokko1<br>rokko2        | HPE ProLiant<br>DL380 Gen10 | Intel Xeon Gold 6248<br>2.5GHz (20cores x2)                                                 | 768GB                        |
| CPUノード<br>(Thin/Fatノード) | node01~56<br>nodef01~08 | HPE Apollo<br>2000 Gen10    | Intel Xeon Gold 6248<br>2.5GHz (20cores x2)                                                 | 192GB (Thin)/<br>768GB (Fat) |
| GPUノード<br>(gpu搭載ノード)    | nodeg01                 | HPE Apollo<br>6500 Gen10    | Intel Xeon Gold 6248<br>2.5GHz (20cores x2) /<br>NVIDIA V100 32GB<br>SXM2 x8                | 768GB                        |
| VEノード<br>(VE搭載ノード)      | nodev01~02              | HPE Apollo<br>6500 Gen10    | Intel Xeon Gold 6248<br>2.5GHz (20cores x2) /<br>NEC Vector Engine<br>Accelerator Module x8 | 768GB                        |
| 共有メモリノード                | kofuji                  | HPE ProLiant<br>DL560 Gen10 | Intel Xeon Gold 6248<br>2.5GHz (20cores x2)<br>x4 (Total 80cores)                           | 3TB                          |

# フロントエンドサーバ(rokko)へのログイン

- アカウントシート: アカウント名と初期パスワード
- フロントエンドサーバ IPアドレス  
rokko1: 172.25.61.33,  
rokko2: 172.25.61.34
- SSHコマンド  
ssh -Y アカウント名@172.25.61.33  
(-YはX11転送を有効にするオプション。GUI操作を可能にする)  
⇒ログインできたらコマンド“xclock”を入力。時計が出てきたらOK。

# 演習ファイルのコピー

演習ファイル: /home/y273s011/share/khpc2026/20260914.tar

- HOMEディレクトリ: ログイン後に最初にいる場所  
/home/アカウント名
- 作業ディレクトリの作成  
mkdir khpc2026
- 演習ファイルのコピー  
cp /home/y273s011/share/2026/20260914.tar ~/khpc2026/  
cd ~/khpc2026/  
tar xvf 20260914.tar

# プログラミング環境設定

- moduleコマンド
  - Module環境のロード／アンロード  
**module\_load\_intel / module\_unload\_intel**  
**module\_load\_gnuplot / module\_unload\_gnuplot**
  - 現在の設定  
**module\_list**
  - 利用可能なmoduleの確認  
**module\_avail**
  - Module環境の初期化  
**module\_purge**
- 「利用者マニュアル\_rev1.4」の12頁参照

# プログラミング環境

- コンパイラ(「利用者マニュアル」30頁以降)
  - Intel  
`icx_ -O3_ hello.c_ -o_ hello.out`  
オプション                      実行ファイルの指定
- 実行はPBSジョブ管理システムを使います.  
(「利用者マニュアル」14頁)
  - ジョブスクリプトの利用  
`qsub_ <ジョブスクリプト>`
  - インタラクティブジョブ  
`qsub_-l`

# PBSジョブ管理システム

- キュー構成

| キュー名     |   | ジョブのコア数制限 | 最長実行時間 | 使用できるノード            | アクセラレータ数 | 優先度 | インタラクティブ |
|----------|---|-----------|--------|---------------------|----------|-----|----------|
| T        |   | 1-40      | 30min  | CPU ノード (Thin)      | -        | 150 | 可        |
| TF       |   | 1-40      | 30min  | CPU ノード (Fat)       | -        | 150 | 可        |
| C        | S | 1-80      | 12h    | CPU ノード (Thin)      | -        | 130 | 可        |
|          | M | 81-200    | 24h    | CPU ノード (Thin)      | -        | 90  | -        |
|          | L | 201-2240  | 24h    | CPU ノード (Thin)      | -        | 70  | -        |
|          | H | 2241-2560 | 24h    | CPU ノード (Thin, Fat) | -        | 50  | -        |
| SF       |   | 1-80      | 12h    | CPU ノード (Fat)       | -        | 130 | 可        |
| MF       |   | 81-160    | 24h    | CPU ノード (Fat)       | -        | 90  | -        |
| LF       |   | 161-320   | 24h    | CPU ノード (Fat)       | -        | 70  | -        |
| LONG     |   | 1-40      | 168h   | CPU ノード (Thin)      | -        | 130 | -        |
| G        |   | 1-40      | 24h    | GPU 搭載ノード           | GPU 8    | 130 | 可        |
| V        |   | 1-40      | 24h    | VE 搭載ノード            | VE 8     | 130 | 可        |
| VL       |   | 41-80     | 24h    | VE 搭載ノード            | VE 16    | 70  | -        |
| SMP      |   | 1-80      | 24h    | 共有メモリノード            | -        | 130 | 可        |
| SMP-LONG |   | 1-80      | 168h   | 共有メモリノード            | -        | 130 | -        |

サマースクールの期間中限定  
WSキュー: 15 CPUノード

- キュー名 : ジョブ投入の際に指定するキュー名です。
- ノード/コア数制限 : ジョブ投入の際に予約できる最小・最大ノード/コア数です。
- 最長実行時間 : ジョブ投入の際に予約できる最大実行時間です。
- 使用できるノード : 各キューで利用可能なノードです。
- アクセラレータ数 : 各キューで利用可能なアクセラレータ数です。
- 優先度 : キューの優先順位. 大きい値の方が優先度が高くなります。

# PBSジョブ管理システム

- バッチスクリプトの作成 : **run.sh**

```
#!/bin/bash
#PBS_-q_WS
#PBS_-l_select=1:ncpus=1
#PBS_-N_job_hello
#PBS_-j_oe
```

```
module load intel
cd_${PBS_O_WORKDIR}
```

```
icx_hello.c_-o_hello.out
./hello.out
```

- キューの指定
- 計算リソースの指定
- 任意のジョブ名の指定
- 標準出力と標準エラー出力を統合

- module環境設定
- 投入ディレクトリに移動

- コンパイル
- 実行

- ジョブ投入: **qsub\_run.sh**

# PBSジョブ管理システム

- 結果表示

**cat\_ job\_hello.o<ジョブID>**

- 実行ジョブの確認（「利用者マニュアル」28頁）

**qstat**

表 5 qstat コマンドの主なオプション

| オプション          | 説明                                                        |
|----------------|-----------------------------------------------------------|
| -u_<user_name> | 指定したユーザのジョブの状態を表示します。                                     |
| -f_<ジョブ ID>    | 特定のジョブの詳細な状況を表示します。ジョブ ID を省略すると、すべてのジョブについての詳細な状況を表示します。 |
| -Q             | キューの状況を表示します。                                             |
| -s             | ジョブコメントおよびその他の情報が表示されます。キューイング中のジョブは実行待ちになっている理由が表示されます。  |

- ジョブのキャンセル（「利用者マニュアル」29頁）

**qdel\_<ジョブID>**

# PBSジョブ管理システム

- インタラクティブキュー
  - **qsub -l -q WS** クラスタノードにログイン.  
“-q”で利用するキューの指定.
  - Job IDが割り当てられる. **qstat**で確認.
  - 作業ディレクトリに移動して作業を実行  
**module\_load\_intel**  
**icx\_hello.c**  
**./a.out**
  - **exit** コマンドで終了. (ノードからログアウト. )  
**必ずexitでログアウトして下さい!!**

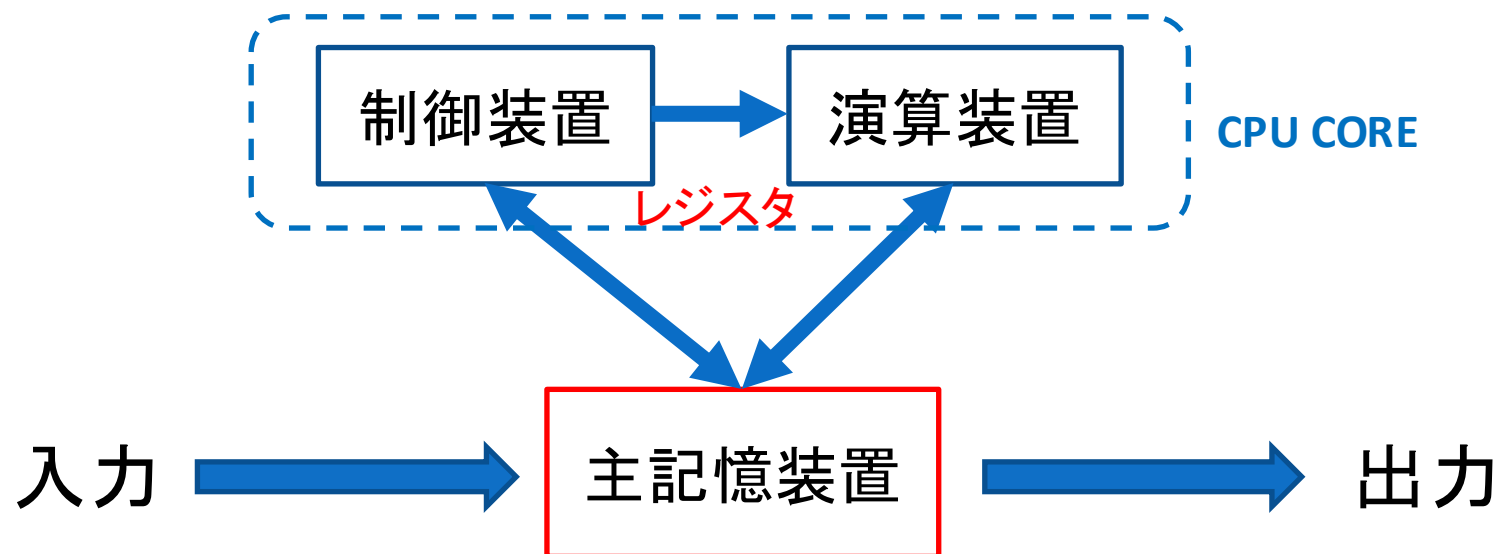
# 演習0(hello.c)

- バッチスクリプト(run.sh)を使って実行する.
- インタラクティブキューを使って実行する.
  1. qsub -l -q WS
  2. cd 作業ディレクトリ
  3. icx hello.c -o hello.out
  4. ./hello.out

※ Linux環境構築やネットワーク設定等については補足資料を参照すること.

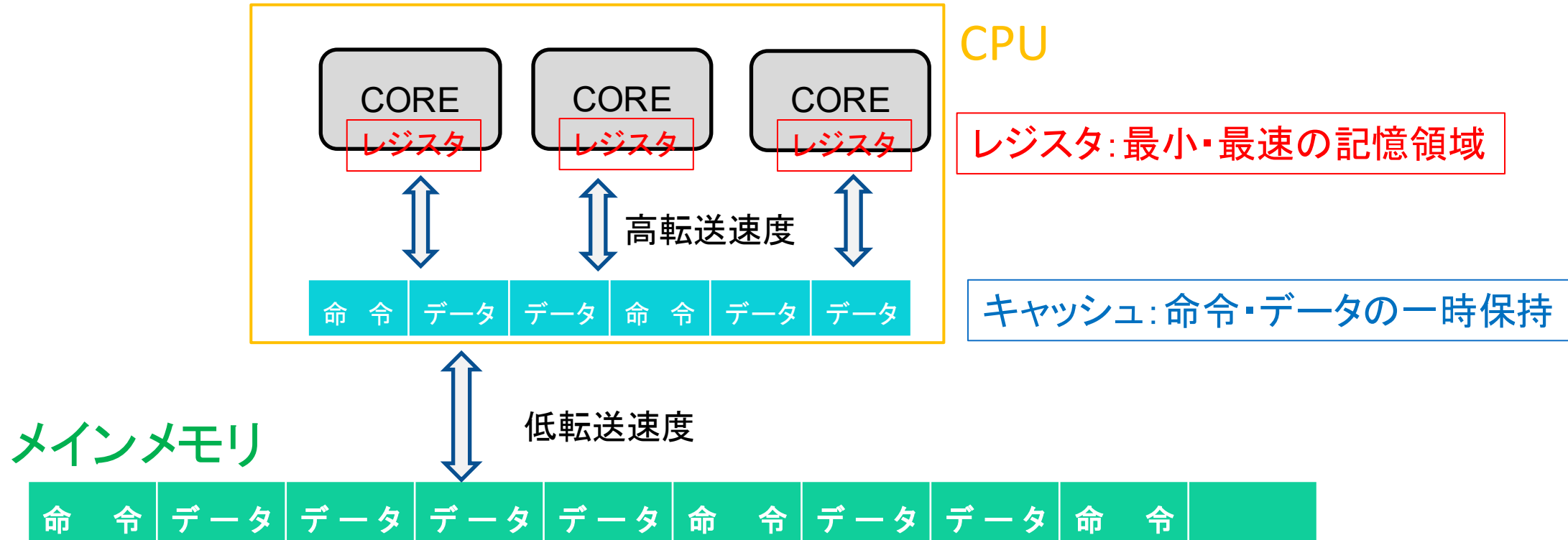
# 逐次プログラムの高速化

# 計算機の基本構造



1. 主記憶装置(メモリ), **1次元のアドレス空間**をもつ.
2. 命令とデータがともに主記憶に置かれる.
3. CPUが命令を読み取り、解釈し、演算装置で処理する.

# メモリ階層



主記憶装置(メモリ): 大容量, 連続の番号を持ったアドレス空間

# 逐次計算の高速化

## ● ループ展開

- 繰り返し処理で毎回発生する終了条件のチェックの低減
- ループ制御のカウンタやポインタの更新回数の低減

行列積の計算

```
for(i=0;i<IMAX;i++)  
  for(j=0;j<IMAX;j++)  
    for(k=0;k<IMAX;k++)  
      c[i][j]+=a[i][k]*b[k][j];
```

2つ目のループについて展開

```
for(i=0;i<IMAX;i++)  
  for(j=0;j<IMAX;j+=8)  
    for(k=0;k<IMAX;k++){  
      c[i][j]+=a[i][k]*b[k][j];  
      c[i][j+1]+=a[i][k]*b[k][j+1];  
      c[i][j+2]+=a[i][k]*b[k][j+2];  
      c[i][j+3]+=a[i][k]*b[k][j+3];  
      c[i][j+4]+=a[i][k]*b[k][j+4];  
      c[i][j+5]+=a[i][k]*b[k][j+5];  
      c[i][j+6]+=a[i][k]*b[k][j+6];  
      c[i][j+7]+=a[i][k]*b[k][j+7];  
    }
```

# 逐次計算の高速化

## ● 一時変数の活用

- ループ内で同じ変数へのアクセスが多い場合、一時変数を用いると冗長な命令を省く事ができる.

```
for(i=0;i<IMAX;i++)
  for(j=0;j<IMAX;j+=8)
    for(k=0;k<IMAX;k++){
      c[i][j]+=a[i][k]*b[k][j];
      c[i][j+1]+=a[i][k]*b[k][j+1];
      c[i][j+2]+=a[i][k]*b[k][j+2];
      c[i][j+3]+=a[i][k]*b[k][j+3];
      c[i][j+4]+=a[i][k]*b[k][j+4];
      c[i][j+5]+=a[i][k]*b[k][j+5];
      c[i][j+6]+=a[i][k]*b[k][j+6];
      c[i][j+7]+=a[i][k]*b[k][j+7];
    }
```

配列a[i][k]へ冗長なアクセス

```
for(i=0;i<IMAX;i++)
  for(j=0;j<IMAX;j+=8)
    for(k=0;k<IMAX;k++){
      t=a[i][k];
      c[i][j]+=t*b[k][j];
      c[i][j+1]+=t*b[k][j+1];
      c[i][j+2]+=t*b[k][j+2];
      c[i][j+3]+=t*b[k][j+3];
      c[i][j+4]+=t*b[k][j+4];
      c[i][j+5]+=t*b[k][j+5];
      c[i][j+6]+=t*b[k][j+6];
      c[i][j+7]+=t*b[k][j+7];
    }
```

# 演習1 (ex01.c)

- ループ展開無し, ループ展開で2回処理, 4回処理, 8回処理の場合について実行速度を比較せよ.
- ループ展開で8回処理した後に, さらに一時変数を用いることで実行速度が速くなるか確認せよ.

※ここでは手動による高速化の効果を確認するためコンパイルオプションを"-O0"(最適化無し)としてコンパイルする.

# 逐次計算の高速化

## ● 水平参照と垂直参照

- 水平参照の方がキャッシュミスが少ない.

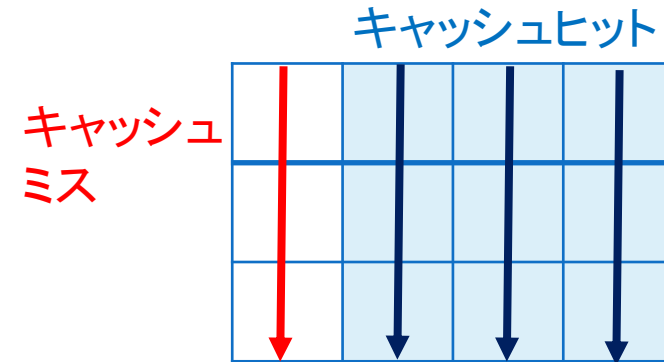
2次元配列  $A[i][j]$      $A[i][j]$ は1次元メモリ空間で  $i * J + j$  番目のデータ.

|   |       |       |          |       |       |       |       |       |
|---|-------|-------|----------|-------|-------|-------|-------|-------|
| 0 | 1     | 2     | 3        | ..... | J - 4 | J - 3 | J - 2 | J - 1 |
| J | J + 1 | J + 2 | J + 3    | ..... | 2J-4  | 2J-3  | 2J-2  | 2J-1  |
| ⋮ |       |       |          | ⋮     |       |       |       |       |
|   |       |       | $[i][j]$ |       |       |       |       |       |
|   |       |       |          |       |       |       |       |       |
|   |       |       |          |       |       |       |       |       |
|   |       |       |          |       |       |       |       |       |

$A[i][j]$ のデータが必要な場合, その前後のデータをまとめキャッシュにあげる.

# 逐次計算の高速化

- サブブロック分割



|   |       |       |       |       |        |        |        |        |
|---|-------|-------|-------|-------|--------|--------|--------|--------|
| 0 | 1     | 2     | 3     | ..... | J - 4  | J - 3  | J - 2  | J - 1  |
| J | J + 1 | J + 2 | J + 3 | ..... | 2J - 4 | 2J - 3 | 2J - 2 | 2J - 1 |
| ⋮ |       |       |       | ⋮     |        |        |        |        |
|   |       |       |       |       |        |        |        |        |
|   |       |       |       |       |        |        |        |        |
|   |       |       |       |       |        |        |        |        |
|   |       |       |       |       |        |        |        |        |
|   |       |       |       |       |        |        |        |        |

# 演習2

- 2つの高速化処理についてベンチマーク実行.  
(コンパイルの最適化オプションは“-O0”を指定)
  - a. 垂直参照と水平参照の比較  
ex02a.cを水平参照のコードに改良せよ.
  - b. 垂直参照とサブブロック分割の比較.  
ex02b.cでoffsetのサイズを変更して比較せよ.  
offset=4, 8, 16, 32, 64, 128

# ベクトル化

- SIMD (Single Instruction Multiple Data)
  - 1命令で複数要素の演算を行う.

```
for(i=0;i<n;i++){  
    C[i]=A[i]+B[i];  
}
```

- [SSE]: float4要素を一度に計算
- [AVX]: float8要素
- [AVX-512]: float 16要素

|       |      |      |      |      |
|-------|------|------|------|------|
|       | A[0] | A[1] | A[2] | A[3] |
| +     | B[0] | B[1] | B[2] | B[3] |
| <hr/> |      |      |      |      |
|       | C[0] | C[1] | C[2] | C[3] |

# コンパイラーによるベクトル化

- ループの自動ベクトル化(-xまたは-axオプション)
  - 最適化レポート(-qopt-reportオプション)でベクトル化状況を確認

```
icx -xhost ex03.c -qopt-report -o ex03.out
```

- -xhost: コンパイルを行うホスト・プロセッサで利用可能な最上位の命令セット向けのコードを生成.
- ex03.optrptにベクトル化状況を報告.

## 演習3(ex03.c)

- コンパイル時にSIMD命令オプション (-xsse4.2, -xavx2, -xcore-avx512)を有効にした場合と、ベクトル化を無効にした場合(-no-vec)について計算速度を比較せよ.

# 熱伝達問題の差分計算

# 連立一次方程式の解法

- ヤコビの反復法 (Jacobi method)

$$A\mathbf{x} = \mathbf{b}$$

$$A = D + U + L$$

$D$ : diagonal matrix,

$U$ : upper triangular matrix:

$L$ : lower triangular matrix

$$\mathbf{x} = D^{-1} (\mathbf{b} - (U + L)\mathbf{x})$$

$$\mathbf{x}^{(k+1)} = D^{-1} (\mathbf{b} - (U + L)\mathbf{x}^{(k)})$$

$$\mathbf{x}^{(k)} \rightarrow \mathbf{x}^*$$

- ヤコビの反復法の形式

$$Ax = b$$

$$A = D + U + L$$

$D$ : diagonal matrix,

$U$ : upper triangular matrix:

$L$ : lower triangular matrix

$$x = D^{-1} (b - (U + L)x)$$

$$x_i = c_0 x_0 + c_1 x_1 + \dots + c_{i-1} x_{i-1} + c_{i+1} x_{i+1} + \dots + c_N x_N$$

1.  $x$  の  $i$  成分について, 成分  $i$  以外のベクトル要素の線形結合で表す.
2.  $i=0, \dots, N$  の  $N+1$  個の連立一次方程式を反復法で解く.

- ヤコビ反復法 (3 × 3行列)

$$\begin{pmatrix} 5 & 2 & -1 \\ 1 & 3 & 1 \\ 1 & -1 & 5 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 5 \\ -2 \\ 4 \end{pmatrix} \quad \text{非対角成分を除いて} \\ \text{右辺に移行}$$

$$\begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 5 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 5 \\ -2 \\ 4 \end{pmatrix} - \begin{pmatrix} 0 & 2 & -1 \\ 1 & 0 & 1 \\ 1 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \frac{1}{5} & 0 & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{5} \end{pmatrix} \left[ \begin{pmatrix} 5 \\ -2 \\ 4 \end{pmatrix} - \begin{pmatrix} 0 & 2 & -1 \\ 1 & 0 & 1 \\ 1 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \right]$$

- ヤコビの反復法(3×3行列)

initial value  $(x^{(0)}, y^{(0)}, z^{(0)}) = (x_0, y_0, z_0)$

適当な初期値を使って  
反復計算を実行.

$$\begin{pmatrix} x^{(k+1)} \\ y^{(k+1)} \\ z^{(k+1)} \end{pmatrix} = \begin{pmatrix} \frac{1}{5} & 0 & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{5} \end{pmatrix} \left[ \begin{pmatrix} 5 \\ -2 \\ 4 \end{pmatrix} - \begin{pmatrix} 0 & 2 & -1 \\ 1 & 0 & 1 \\ 1 & -1 & 0 \end{pmatrix} \begin{pmatrix} x^{(k)} \\ y^{(k)} \\ z^{(k)} \end{pmatrix} \right]$$

$$\begin{pmatrix} x^{(k)} \\ y^{(k)} \\ z^{(k)} \end{pmatrix} \rightarrow \begin{pmatrix} x^* \\ y^* \\ z^* \end{pmatrix} \quad (k \rightarrow \infty)$$

解が収束するまで反復する.

(x,y,z)が収束する場合には、その値は連立一次方程式の解を与える.

- ヤコビの反復法 (3 × 3行列)
  - 収束判定の例

$$\begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} = \begin{pmatrix} 5 & 2 & -1 \\ 1 & 3 & 1 \\ 1 & -1 & 5 \end{pmatrix} \begin{pmatrix} x^{(k)} \\ y^{(k)} \\ z^{(k)} \end{pmatrix} - \begin{pmatrix} 5 \\ -2 \\ 4 \end{pmatrix}$$

$$\sqrt{e_x^2 + e_y^2 + e_z^2} \ll 1$$

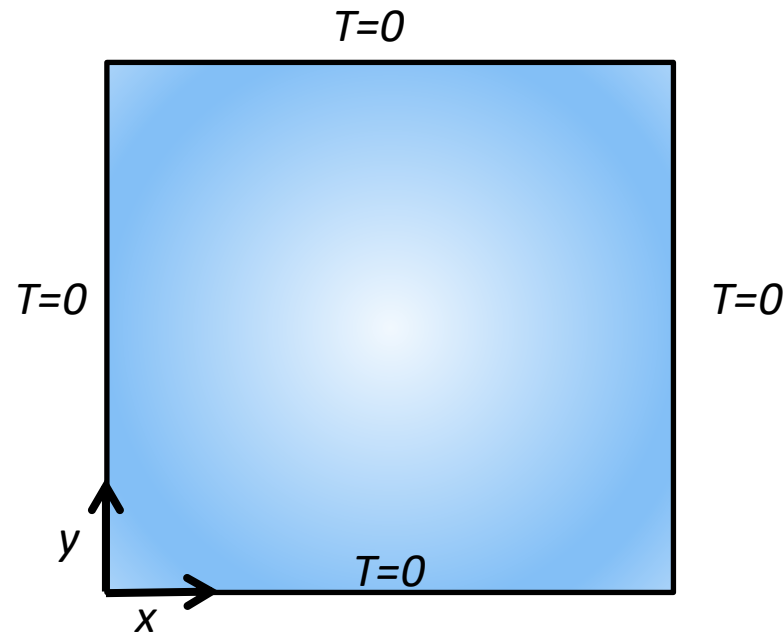
元の方程式との残差を求めて  
十分小さくなるまで反復する.

サンプルコード `yacobi_3by3.c`

# 熱伝達問題の差分解法

- 温度場を計算する。(ポアソン方程式)

$$-\Delta T = -\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) = 10$$



- 温度場を計算する。(ポアソン方程式)

$$-\Delta T = -\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) = 10$$

- 差分化する。

$$\begin{aligned} T(x_i, y_j) &= T_{i,j} , \\ x_i = y_i &= \frac{i}{N} \quad (i = 0, \dots, N), \\ \Delta h &= 1/N \end{aligned}$$

- 温度場を計算する。(ポアソン方程式)

$$-\Delta T = -\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) = 10$$

- 差分化する。

$$T_{i,j} = \frac{1}{4} (10\Delta h^2 + T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1})$$
$$(i, j = 1, \dots, N - 1)$$

- $T_{i,j}$  の線形連立方程式

$$\left. \begin{aligned} T_{0,j} &= T_{N,j} = 0, (j = 0, \dots, N) \\ T_{i,0} &= T_{i,N} = 0, (i = 0, \dots, N) \end{aligned} \right\} \begin{array}{l} \text{境界条件} \\ \text{として固定} \end{array}$$

## ヤコビの反復法

$$\begin{aligned} T_{i,j}^{(k+1)} &= \frac{1}{4} \left( 10\Delta h^2 + T_{i-1,j}^{(k)} + T_{i+1,j}^{(k)} + T_{i,j-1}^{(k)} + T_{i,j+1}^{(k)} \right) \\ (i, j &= 1, \dots, N-1) \end{aligned}$$

サンプルコード poisson.c

# 演習4

- 熱伝達問題のコードを使って以下を実施せよ.
  - SIMD命令の効果の確認.  
ベクトル化を有効にした場合(-xsse4.2, -xavx2, -xcore-avx512)と無効にした場合(-no-vec)について計算速度を比較せよ.
  - Gnuplotによる温度分布の可視化.  
gnuplot  
>splot "poisson.dat"  
>set pm3d map  
>replot

# (発展) ガウス・ザイデル法

- 熱伝達問題の差分式を次の反復式によって計算する.

for i=1 to i=N-1

for j=1 to j=N-1

$$T_{i,j}^{(k+1)} = \frac{1}{4} \left( 10\Delta h^2 + T_{i-1,j}^{(k+1)} + T_{i+1,j}^{(k)} + T_{i,j-1}^{(k+1)} + T_{i,j+1}^{(k)} \right)$$

- 反復計算 (poisson.cの43行目) の右辺を  
Told  $\rightarrow$  T  
に変更する.
- 反復計算の収束速度はi,jの走査方向に依存する.

サンプルコード poisson\_gs.c

# (発展) ガウス・ザイデル法

- ヤコビ法に比べて収束が早い.

- 後方依存関係がある.

$$T_{i,j}^{(k+1)} \leftarrow T_{i-1,j}^{(k+1)}, T_{i,j-1}^{(k+1)}$$

- 依存関係がベクトル化を阻害していることが確認できる.

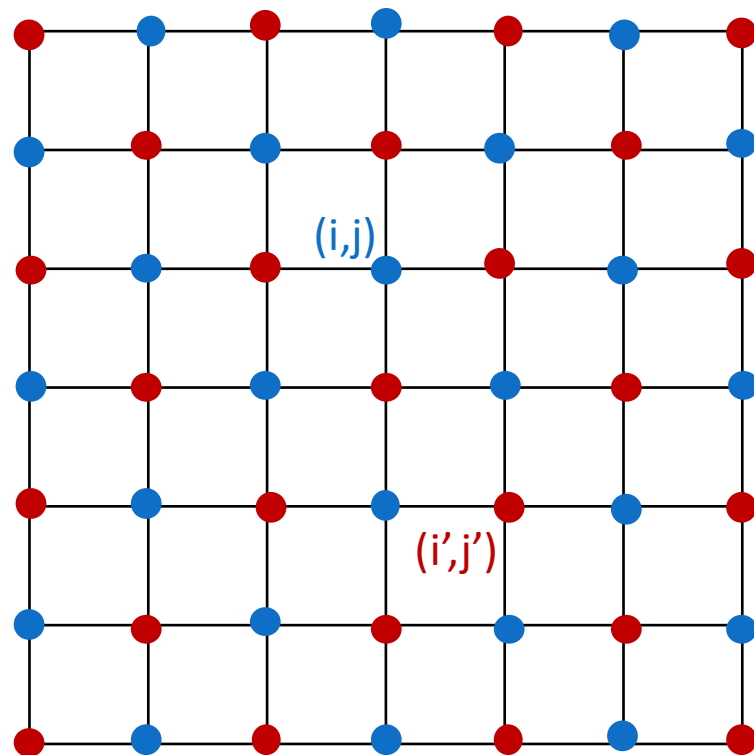
`icx -xhost poisson_gs.c -qopt-report`

- 後方依存の関係を解消する方法を考える.

# (発展) ガウス・ザイデル法

- 前方・後方依存の解消 (計算順序の組み替え)

$$T_{i,j}^{(k+1)} = \frac{1}{4} \left( 10\Delta h^2 + T_{i-1,j}^{(k+1)} + T_{i+1,j}^{(k)} + T_{i,j-1}^{(k+1)} + T_{i,j+1}^{(k)} \right)$$



- はじめに青を計算する.

$$T_{i,j}^{(k+1)} \leftarrow T_{i\pm 1,j}^{(k)}, T_{i,j\pm 1}^{(k)}$$

- 次に赤を計算する.

$$T_{i',j'}^{(k+1)} \leftarrow T_{i'\pm 1,j'}^{(k+1)}, T_{i',j'\pm 1}^{(k+1)}$$

# 自習問題

- 前のページで解説した計算順序の入替を実装し、修正版ガウス・ザイデル法のコードを作成しなさい.
- 元の方法と修正版の方法について、収束速度, 計算速度, ベクトル化の観点で比較しなさい.