スーパーコンピュータ超入門講習会

九州大学情報基盤研究開発センター

皆さんは、ふだん、
どのくらい計算機（コンピュータ）を使っていますか？

- 週一回？
- 日に二～三回？
- 四六時中？

そう、スマートフォンもゲーム機も計算機です。

ふだん、あんまり計算をしているようには見えませんが。

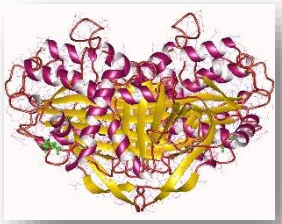
本来、計算機とは、計算をする道具です。

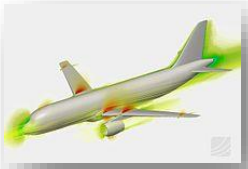
皆さんの代わりに、皆さんよりはるかに速く、
計算します。

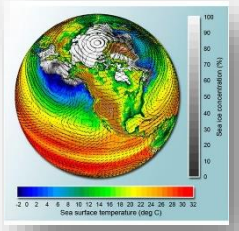
例えばスマートフォンでも、一秒間に数百億回の
計算が出来ます。

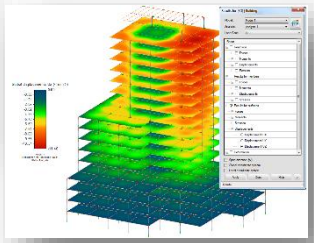
この計算能力を、いろいろな計算に使えます。

例えば、
・飛行機の設計、津波の被害予測、気象予報
　のような、シミュレーションや、
・Alpha Go、自動翻訳、自動運転、株価予測
　のような、機械学習、統計処理









計算機ごとに、計算能力の限界があります。

一般に、スーパーコンピュータというのは、
スマートフォンやPCよりはるかに高い能力を持つ
計算機です。

出典：http://eng.mod.gov.cn/Database/Academies/2013-06/18/content_4455818_2.htm

出典：https://www.llnl.gov/news/aroundthelab/2012/Jun/ATL-061812_sequoia.html

出典：http://www.aics.riken.jp/jp/k/facility.html

世の中の、ほとんどのスーパーコンピュータは、
インターネットにつながっています。

日本にも、そのようなスーパーコンピュータが
いくつかあり、手順に従って申請すれば誰でも（※）
利用出来るものもあります。

九州大学にも、一つあります。

※ある程度の基準や制限などはあります

日本の大学の共同利用スーパーコンピュータ群

組織	計算機名	CPUコア数	アクセラレータ数	理論演算性能	Linpack性能	Top500順位 (2018年11月)
産総研	ABCI	391,680	4,352	37.2PF	19.9PF	7
理研	K computer	705,024	-	11.3PF	10.5PF	18
北海道大	Fujitsu PRIMERGY	39,760	-	3.1PF	2.0PF	95
東北大	NEC SX-ACE	10,240	-	0.71PF	-	-
東京大、筑波大	Oakforest-PACS	556,104	-	24.9PF	13.6PF	14
筑波大	COMA	7,860	786	1.0PF		
東京大	Reedbush-U Reedbush-H Reedbush-L	15,120 4,320 2,304	- 240 256	0.51PF 1.4PF 1.4PF	- 0.80PF 0.82PF	- - -
東工大	TSUBAME 3.0	15,120	2,160	12.1PF	8.1PF	22
名古屋大	Fujitsu FX100 Fujitsu PRIMERGY	92,160 25,656	- 568	3.2PF 0.72PF	2.9PF	63
京都大	Camphor 2 Laurel 2	133,400 30,600	- -	5.5PF 1.0PF	3.1PF 0.82PF	60 -
大阪大	NEC SX-ACE OCTOPUS	6,144 6,552 (Xeon) + 2,816 (Xeon Phi)	- 148	0.42PF 1.5PF	- -	- -
九州大	ITO	76,608	512	9.9PF	-	37

今日の内容

1. スーパーコンピュータの仕組み
2. スーパーコンピュータの使い方
3. 九州大学のスーパーコンピュータシステムITO紹介
4. スーパーコンピュータの開発競争と将来の展望
5. Q & A

今日の内容

1. スーパーコンピュータの仕組み
2. スーパーコンピュータの使い方
3. 九州大学のスーパーコンピュータシステムITO紹介
4. スーパーコンピュータの開発競争
5. Q & A

計算機の「速さ」とは？

- 計算の速さ
  = ある時間の間に実行できる演算の回数

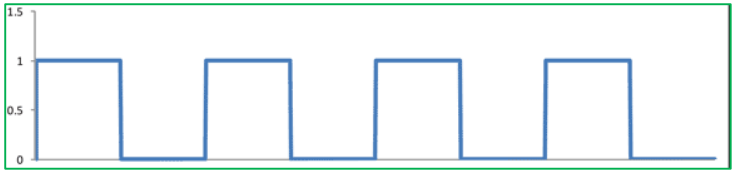
- 最もよく使われる指標：「フロップス」
  FLOPS (FLoating Operations Per Second)
  - floating operation = 実数計算
  - per second = 1秒ごとに

計算機の速さの限界

- 計算機は、以下の P (FLOPS)よりも速く計算することはできない。

$$P = N \times C \times O$$

  - N: その計算機内の CPUの数
  - C: その CPUのクロック周波数（Hz）
    - 一秒間に回路の状態を変える回数（= 命令を実行する回数）



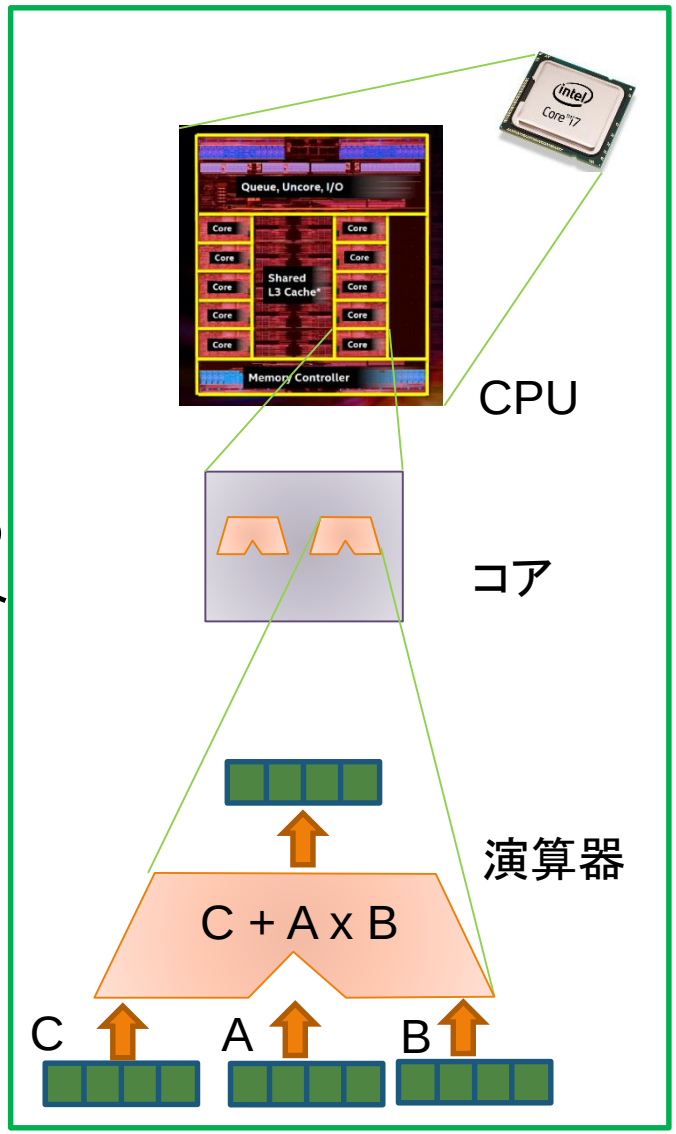
  - O: 一つの CPUが一回のクロックで同時に実行できる演算数

O は、なににによって決まる？

一般的なCPUの構成

- 一つ、または複数の「コア」で構成

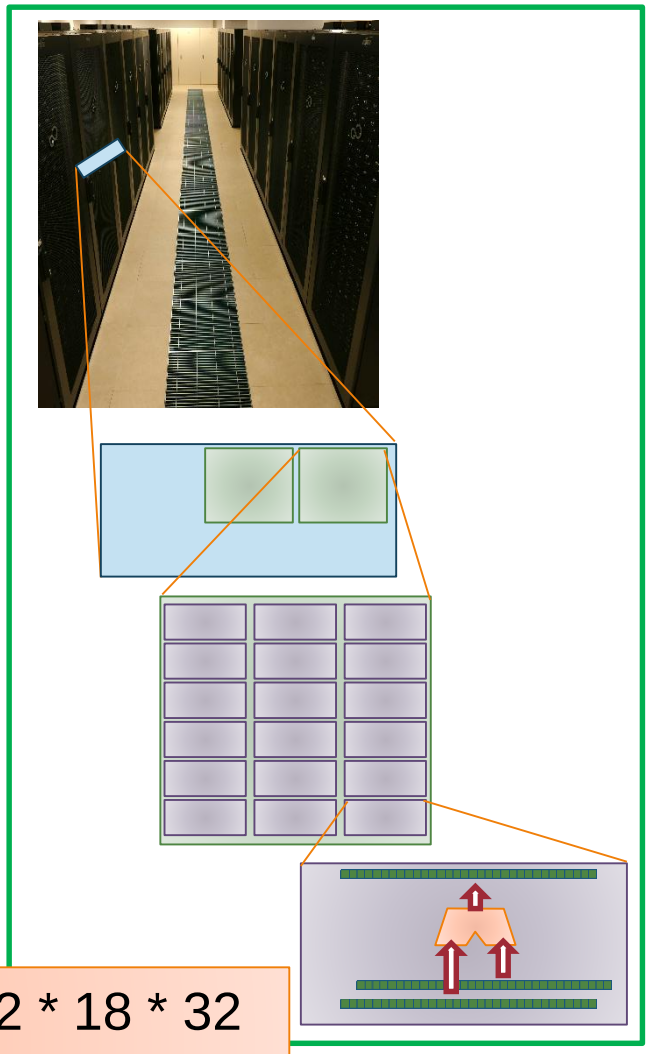
- ほとんどのコアは、一つの命令で
  複数のデータの演算を同時に実行できる
  - SIMD (Single Instruction Multiple Data) 命令
  - さらに、複数の SIMD命令を同時に実行できる
    コアもある
    - 詳細は、CPUのマニュアルを参照
      - 例）Intelの CPUのマニュアル（日本語訳）
        https://www.isus.jp/wp-content/uploads/pdf/64-ia-32-architectures-optimization-manual-April2018-040JA.pdf



O = CPU内のコアの数
　　x コアが一回のクロックで実行できる
　　最大の演算数

例）九州大学のスーパーコンピュータ ITO （サブシステムA）

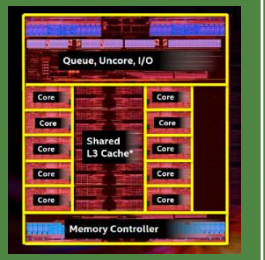
- CPUのクロック周波数：**3.0** GHz
- システムを構成するサーバ数：**2,000** 台
- サーバあたりのCPU数：**2** 個
- CPUあたりのコア数：**18** 個
- コアあたりの最大同時演算数：**32**



6912000 GFLOPS（ギガ フロップス）= 3.0 * 2000 * 2 * 18 * 32
一秒間に 6912兆回の演算が出来る

Kilo = 10^3 , Mega = 10^6 , Giga = 10^9 , Tera = 10^{12} , Peta = 10^{15} , Exa = 10^{18}

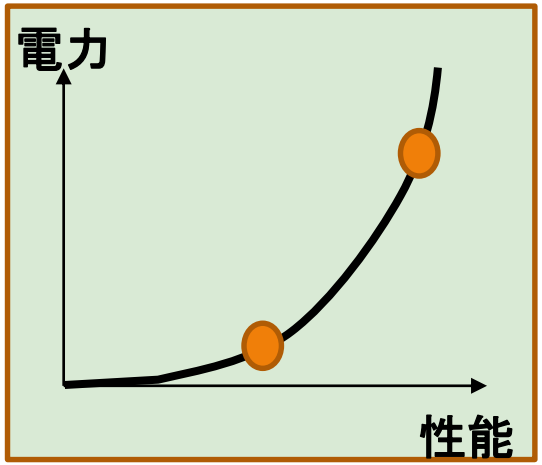
スーパーコンピュータとPC、スマートフォンの演算性能

	スマートフォン Samsung Galaxy S6 (Exynos 7420)	PC (Intel Core i7 6950X, Broadwell)	スーパーコンピュータ ITO　サブシステム A (Intel Xeon Gold 6154, Skylake-SP)	スーパーコンピュータ Summit (IBM Power 9)
				 出典： https://www.ibm.com/thought-leadership/summit-supercomputer/
CPU数	1	1	4,000	9,216
クロック周波数	2.1GHz	3.0GHz	3.0GHz	3.07GHz
コア数	4	10	18	22
コア当たり 最大同時演算数	4	16	32	8
総理論演算性能	33.6 GFLOPS	480 GFLOPS	6,912,000 GFLOPS	4,977,067 GFLOPS

クロック周波数はほぼ同じ ➡ 基本的に CPU数やコア数で演算性能を稼ぐ

なぜ、クロック周波数を上げない？

- クロック周波数で性能を上げる場合：
  性能と電力は比例関係ではない



一般家庭約300世帯分の電力を消費！

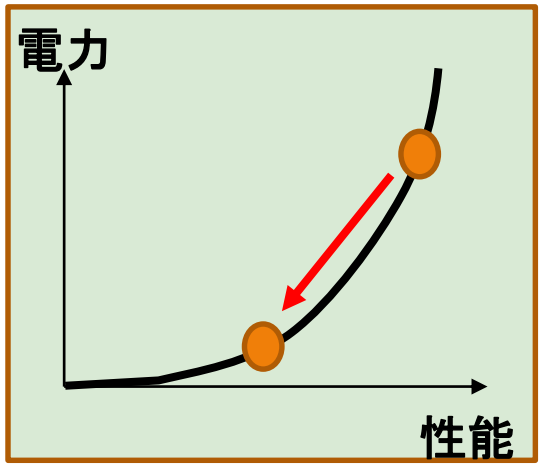
- スーパーコンピュータの消費電力
  - 京コンピュータ（LINPACKベンチマーク計測時）: 12.66 MW
    http://www.fujitsu.com/jp/about/businesspolicy/tech/k/qa/k04.html

電力供給能力、および冷却能力の限界が近づいている

なぜ、CPUやコアを増やす？

- 電力当たりの性能が高いコアを多数並べる

限られた電力でシステム全体の性能向上



例）半分の性能を
1/4 の電力で達成

システム全体の
電力当たり性能 2倍

メニーコア

- 一つのCPU上に電力効率の高いコアを多数配置

- Sunway SW26010 260C: 260 コア
  - Sunway TaihuLightに搭載
  - 中国の独自開発

出典:
http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf

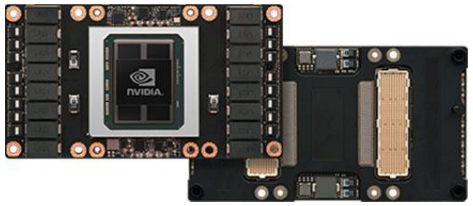
- Intel Xeon Phi : 68〜72 コア
  - Oakforest-PACS (東大 + 筑波大) 等に搭載
  - 過去のアーキテクチャを改良し、最新のSIMD命令を追加

出展:https://software.intel.com/en-us/xeon-phi-apps

アクセラレータ

- コアよりも性能や機能が低い演算装置を多数配置
  - 基本的な演算（積、和など）を高い電力効率で実行
  - ホストCPUから操作

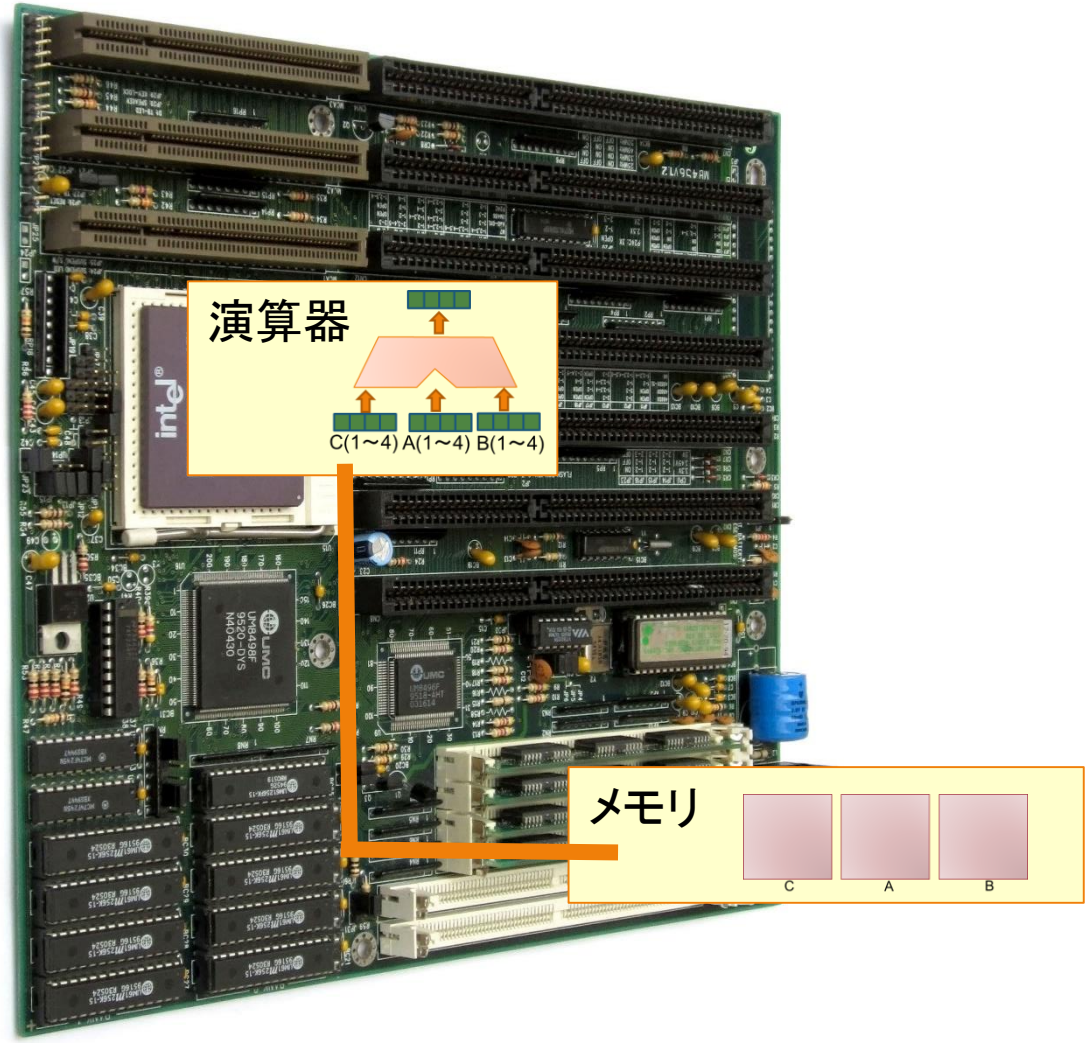
- 例）GPGPU (General Purpose computation on Graphic Processing Unit)
  - グラフィック処理用プロセッサ GPUを
    シミュレーションや機械学習の計算に使用
    - 例) NVIDIA Tesla P100: 3584 CUDAコア



出典：http://www.nvidia.co.jp/object/tesla-p100-jp.html

もう一つの問題：メモリの速さ

- メモリから演算器にデータが届かないと演算が出来ない
- 演算性能とメモリ性能のバランスが重要



メモリの速度が足りない

- 例）ITO サブシステムAのCPUとメモリ
  - CPU: Intel Xeon Gold 6154, 3.0GHz, 18core, 最大32演算
  - メモリ: DDR4

CPUの演算性能	1728 GFLOPS
2演算（Y=A*X+B）当たりのデータ	read 24バイト
演算に必要なメモリ速度	read **20736 GB/秒**
CPUのメモリ速度	**127.8 GB/秒**

CPUの性能を約 0.6% しか利用できない？？

より高速なメモリ

- HMC (Hybrid Memory Cube)
  - Fujitsu PRIMEHPC FX100（京コンピュータ後継機）に搭載
- MCDRAM (Multi Channel DRAM)
  - Intel Xeon Phiの CPUチップ上に搭載
- HBM (High Bandwidth Memory)
  - GPU (NVIDIA Tesla V100 / P100) に搭載

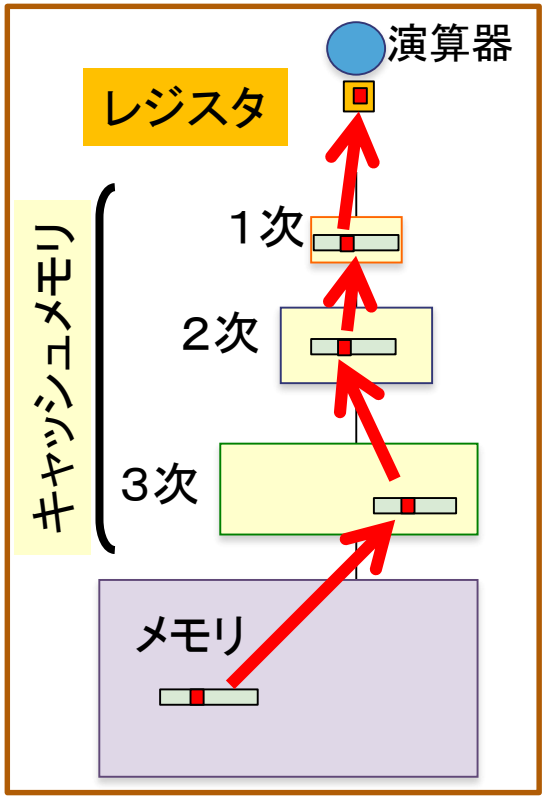
プロセッサと搭載メモリ	最大メモリ速度	最大容量
Intel Xeon Gold, DDR4 2666	127.8GB/秒	1.5TB
Fujitsu SPARC64 XIfx, HMC2	480GB/秒	32GB
Intel Xeon Phi, MCDRAM	400GB/秒	16GB
NVIDIA Tesla P100, HBM2	732GB/秒	16GB

まだ、演算器の演算速度に追いつかない

キャッシュメモリとレジスタ

- キャッシュメモリ:
  演算器とメモリの間に置く高速記憶装置
  - 例）Intel Xeon Gold

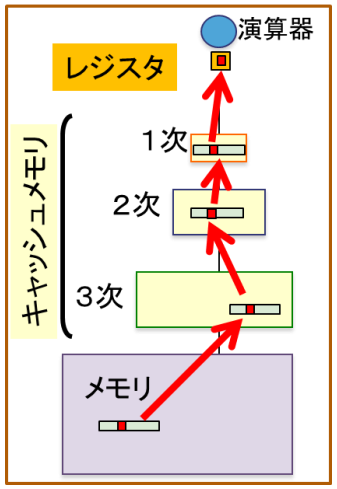
階層	速度	容量
1次	約 128バイト/クロック	32KB / コア
2次	約 32バイト/クロック	256KB / コア
3次	約 16バイト/クロック	36MB / CPU



- レジスタ:
  演算器が直接参照できるデータの置き場所

キャッシュやレジスタのデータをうまく再利用できれば CPUの性能を発揮

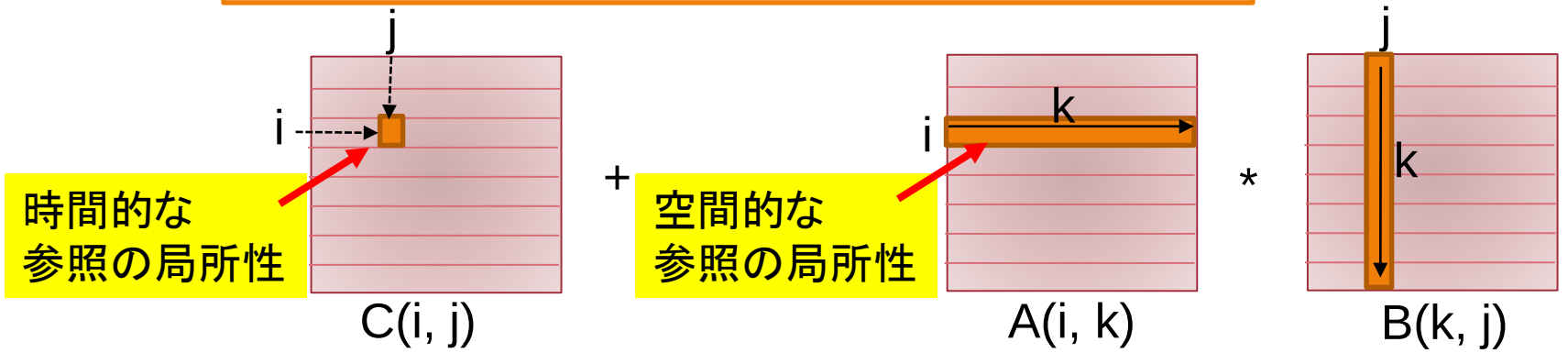
データの再利用＝「参照の局所性」



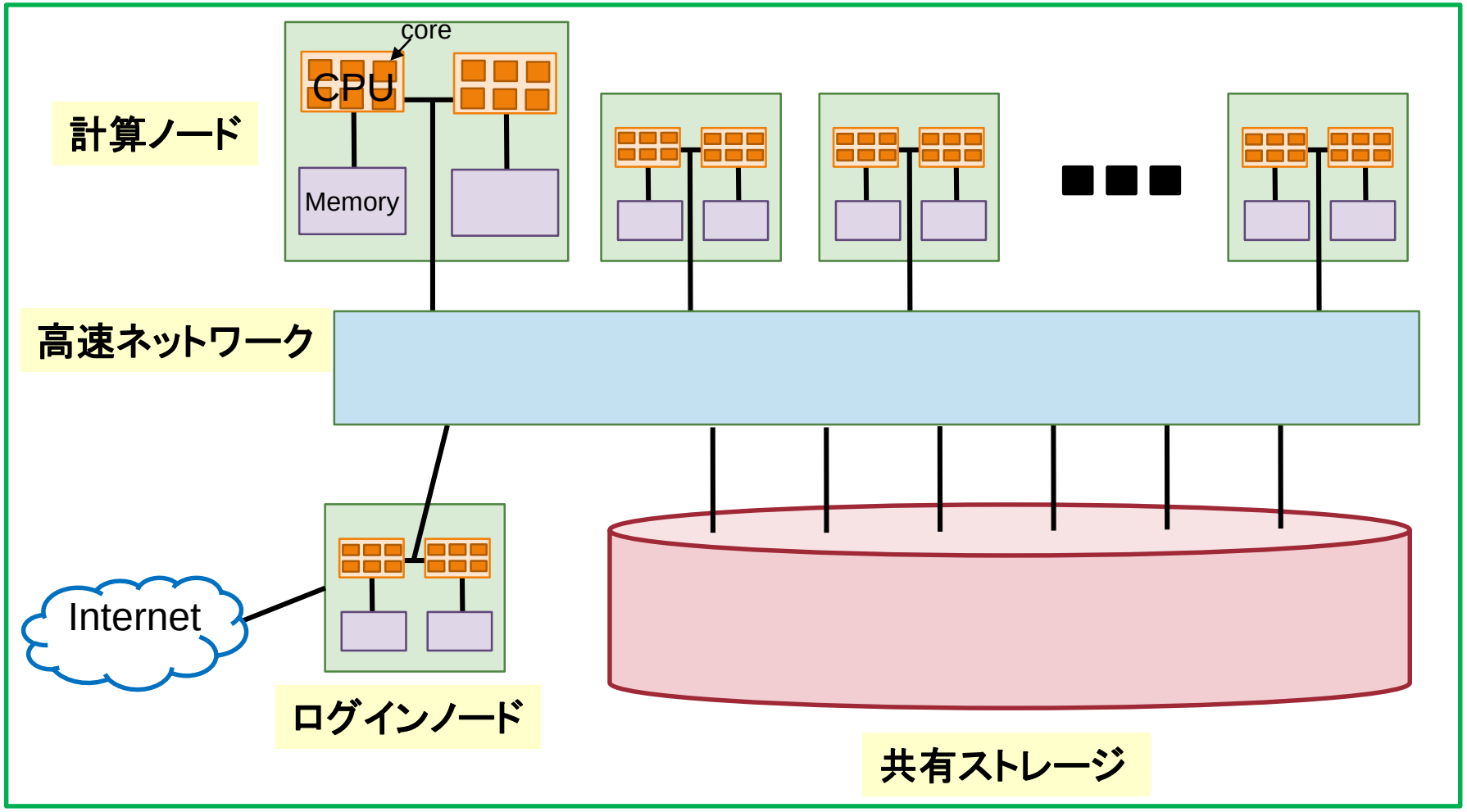
- 時間的な参照の局所性
  - 一度演算に使ったデータを、連続して何度も使う
- 空間的な参照の局所性
  - 一度演算に使ったデータのすぐ近くのデータを連続して使う

- 例）行列同士の積の計算

```
for i = 1 to M
   for j = 1 to M
      for k = 1 to M
         C(i, j) = C(i, j) + A(i, k) * B(k, j)
```

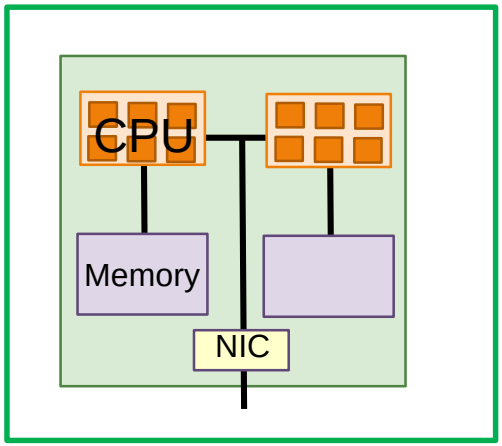


スーパーコンピュータの構成：演算装置とメモリだけではない

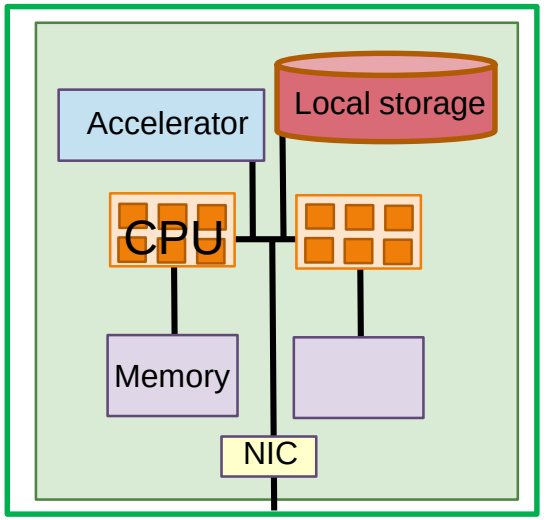


計算ノードの構成

- 必ず載っているもの:
  - 1個～複数個の CPU
  - メモリ
  - ネットワークインタフェース
    等

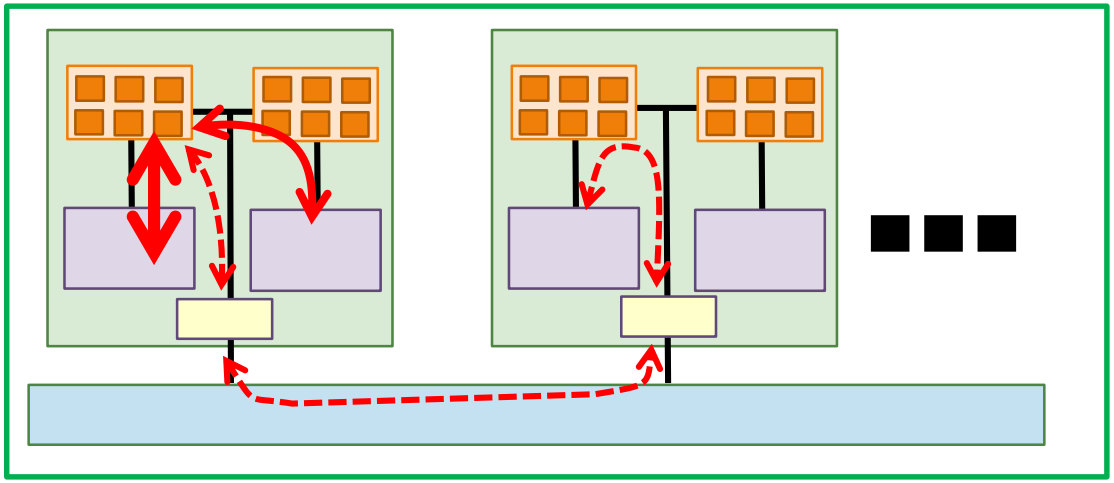


- システムによって載っている場合があるもの:
  - ディスク （HDD or SSD）
    - 用途: OS起動用、一時的なデータ格納用、等
      - 電力、場所、故障率の問題からディスクを搭載せず、
        ネットワーク経由で OSを起動することもある
  - アクセラレータ （GPU等）
    - 用途: 特定の計算を高速化



CPUとメモリの位置関係

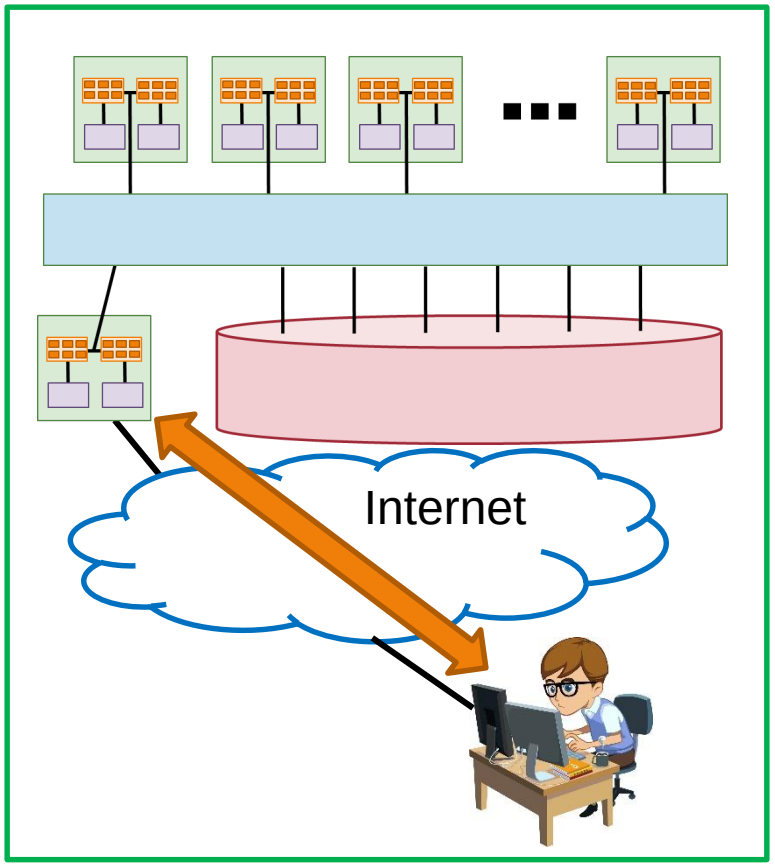
- ノード内の「近い」メモリ
  - 直接読み書き可
  - 高速
- ノード内の「遠い」メモリ
  - 直接読み書き可
  - 若干遅い
- 他ノードのメモリ
  - 直接読み書き不可
    - ネットワーク越しに通信
  - 遅い



計算とデータの配置が性能に大きく影響

ログインノード

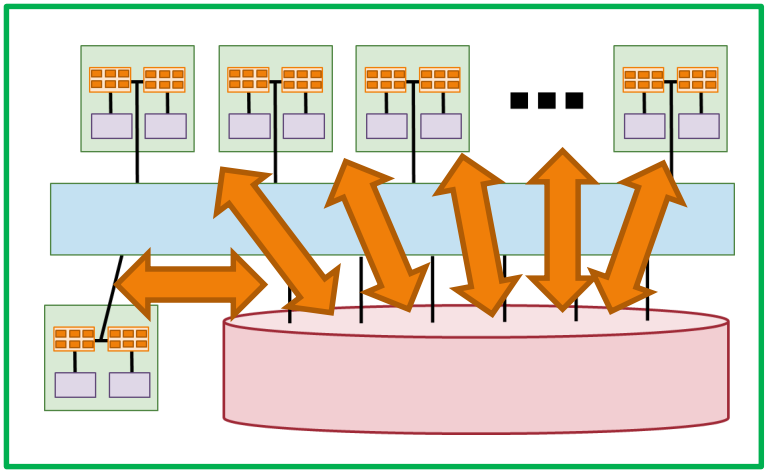
- 外部からログインし、対話的に操作できるノード
  - 計算ノードと別に用意することが多い
    - 計算ノードを計算に専念させるため
    - 計算ノードと CPUや OSが違う場合も
      - 例）京コンピュータ
  - 主な用途
    - 外部とのファイル転送
    - プログラムやデータの準備
    - 計算ノードへの計算依頼
    - 計算結果の処理
      等



共有ストレージ

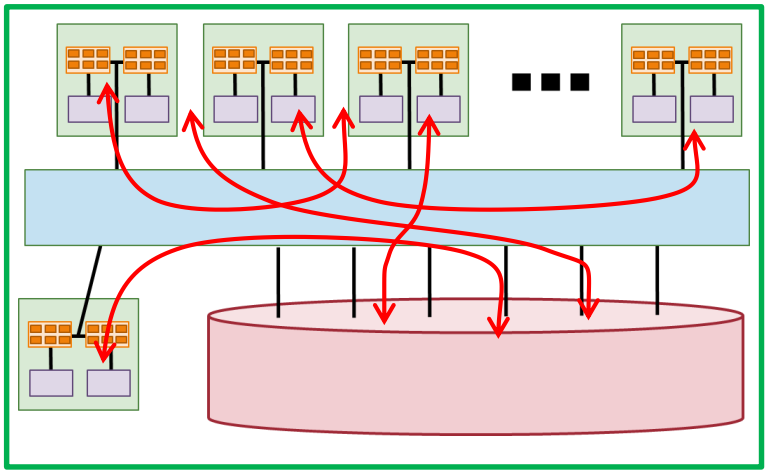
- 計算ノード、ログインノードと、
  複数本の高速ネットワークで接続

- ITOの場合
  - 合計転送速度：**120GB/秒** 程度
  - 容量：**24.6PB**



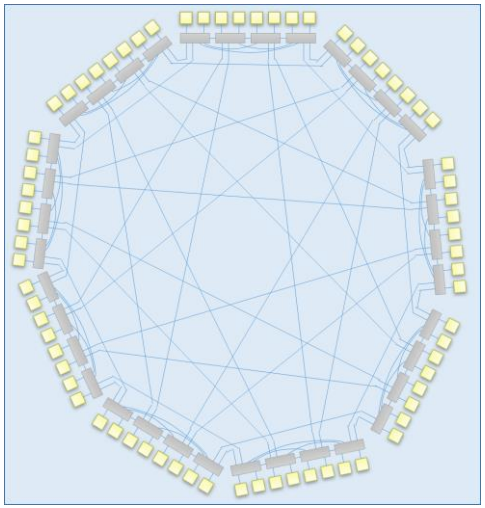
高速ネットワーク

- 通信性能がスーパーコンピュータの性能に大きく影響
  - 複数のノードを使った計算
  - 大量のファイルアクセス

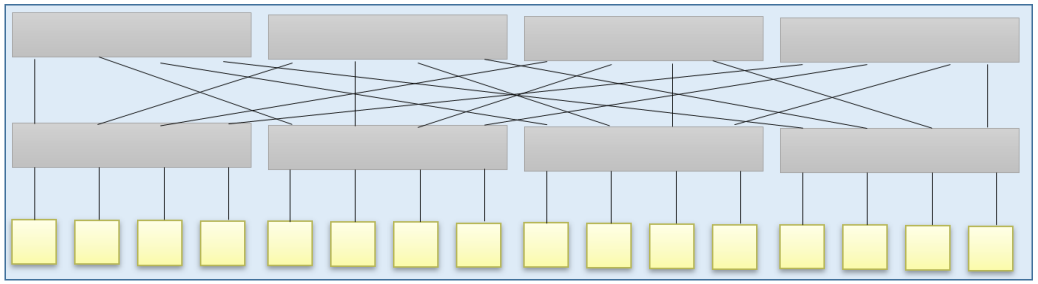


- ITOの場合
  （Mellanox InfiniBand EDR）
  - 最小通信遅延時間: **1μ秒** 程度
    - 一回の通信に最小限必要な時間
      - Ethernet: 数十～数百μ秒
  - 最大通信バンド幅: **12GB/秒** 程度
    - 1秒当たりの最大転送データサイズ
      - Ethernet: 0.1～ 1GB/秒

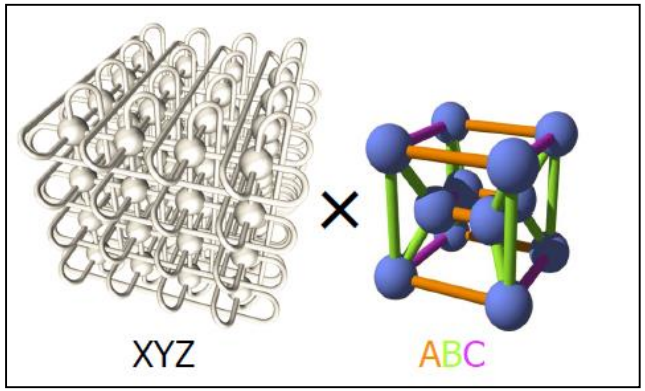
ネットワークの形状(トポロジー)も重要



Dragonfly



Fat Tree



Tofu
(京コンピュータ)

出典: http://www.ssken.gr.jp/MAINSITE/download/newsletter/2011/20110825-sci-1/lecture-5/ppt.pdf

設計目標: なるべく少ない結線数で、互いに干渉せず、より多くの通信を実現

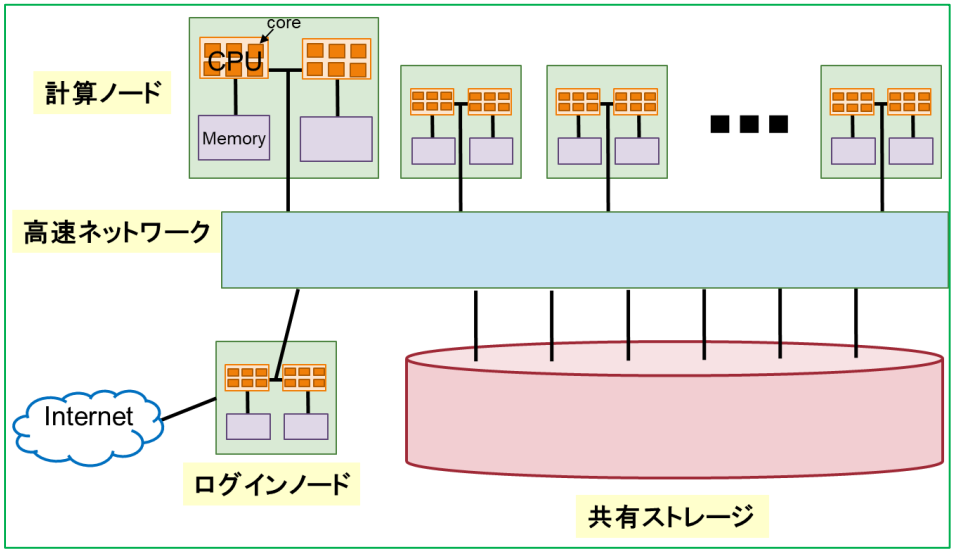
今日の内容

1. スーパーコンピュータの仕組み
2. スーパーコンピュータの使い方
3. 九州大学のスーパーコンピュータシステムITO紹介
4. スーパーコンピュータの開発競争と将来の展望
5. Q & A

スーパーコンピュータの利用手順

1. ログインノードにログイン
2. プログラムやデータの準備
3. 計算ノードに「ジョブ」投入

   （実行完了待ち）
4. 実行結果の処理

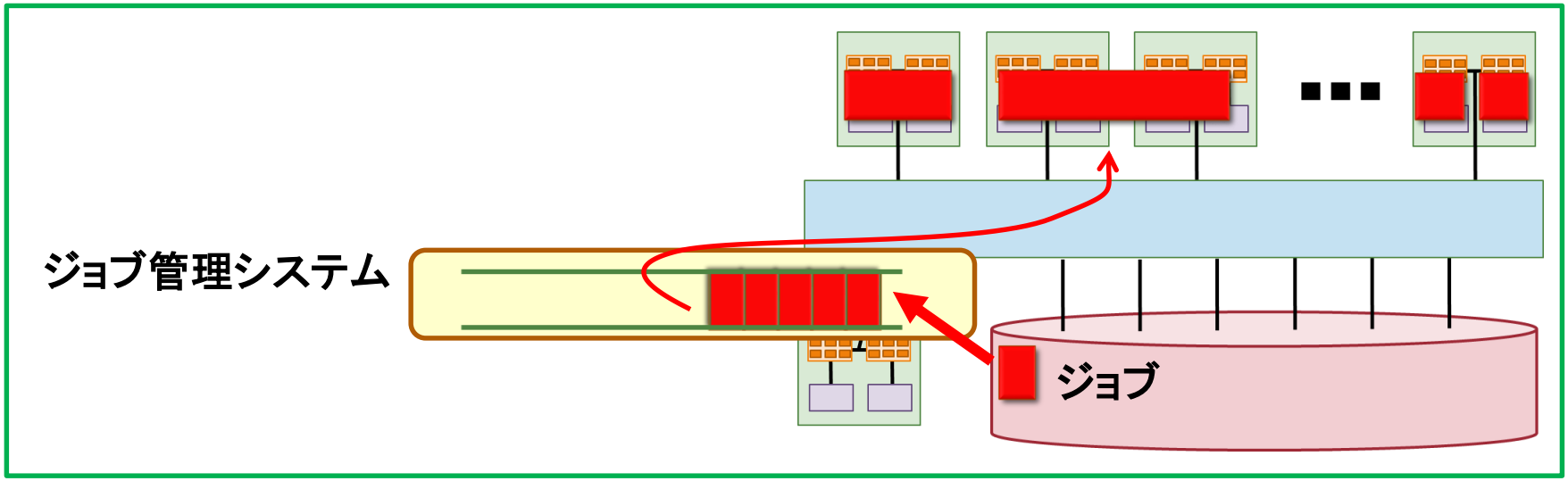


ジョブ

- 計算ノードに実行させるコマンド（を記述したテキストファイル）
  - 条件分岐や繰り返しも記述可能

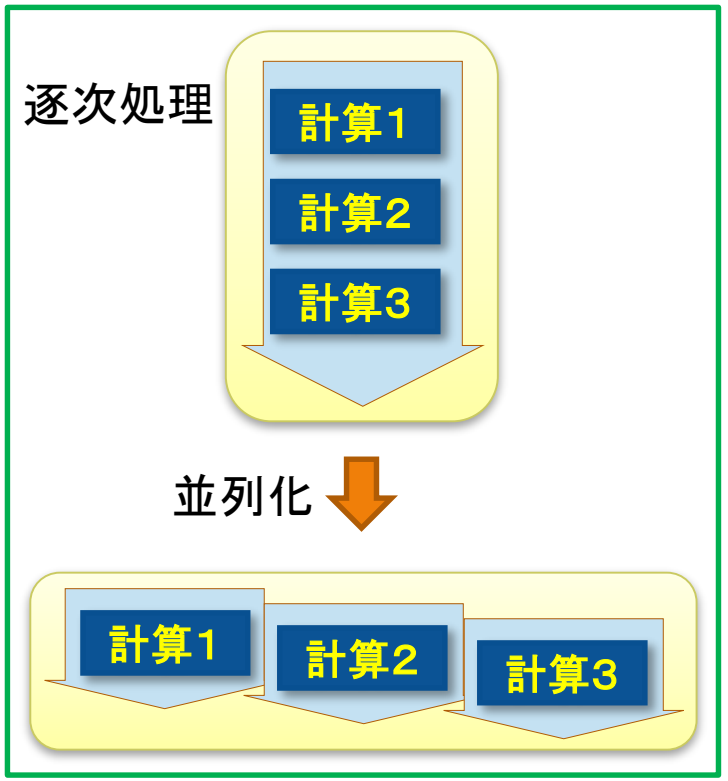
```
for i in data1 data2 data3
do
    ./run  ${i}
done
```

- ジョブ管理システム
  - 計算ノードの空き状況に応じて順にジョブを割り当て

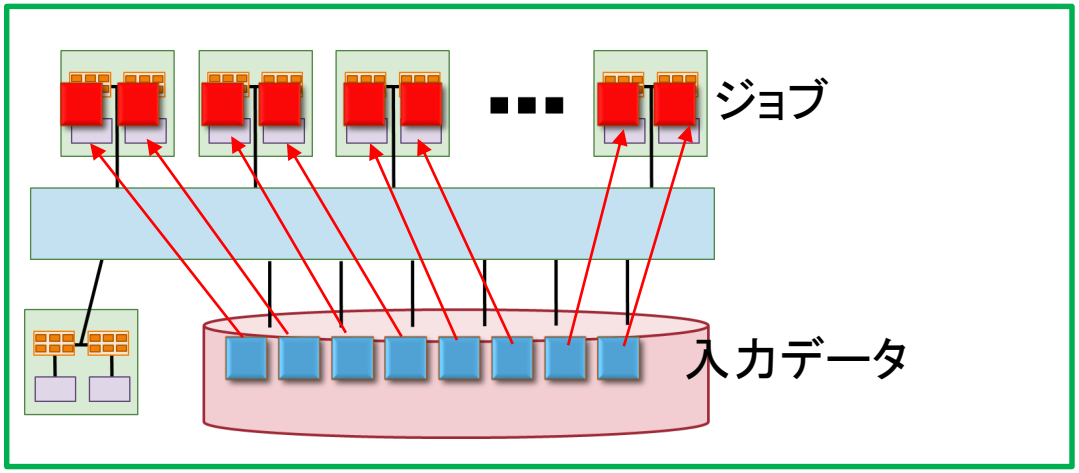


スーパーコンピュータは並列計算機

- スーパーコンピュータの性能を発揮させる使い方：
  計算を分担させ、計算時間を短縮
  - ノード、CPU、コア、演算器
- 逐次処理の並列実行
  - ジョブ単位の並列計算
  - （1つのジョブで多数プログラムを実行）
- プログラムの並列化
  - プロセス並列
  - スレッド並列
  - ハイブリッド並列
  - アクセラレータ、SIMD演算器で並列化



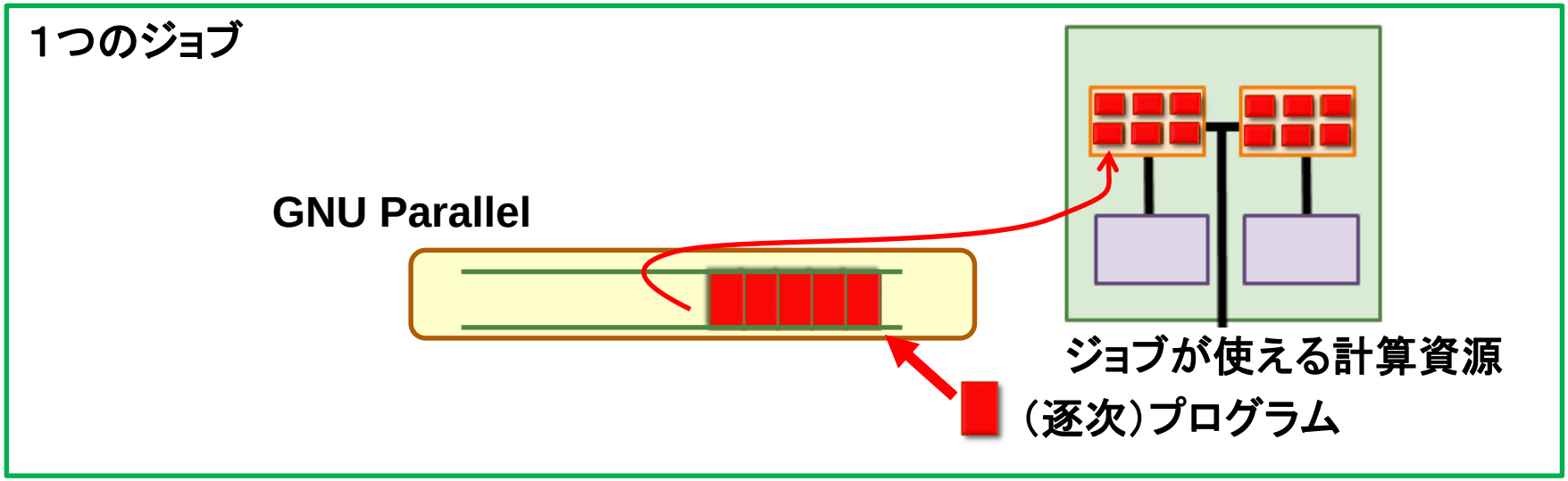
ジョブ単位の並列計算



- 計算したい入力データやプログラムが多数ある場合、簡単にスーパーコンピュータを活用
  - プログラムの改変なし
- 出力ファイルに用心
  - 相互に上書きしないよう、ジョブ毎に出力ファイル名を変える

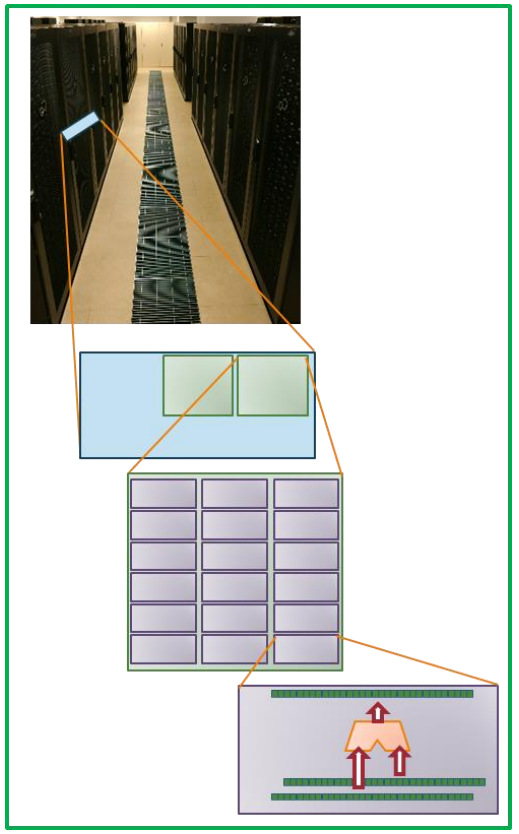
1つのジョブで多数プログラムを実行

- 実際には、ジョブ単位の同時並列数はシステム側で制限
  - 大量ジョブの制御はスーパーコンピュータでも大変・・・
- 解決策：1つのジョブで多数のプログラムを実行
- 便利なツール： GNU Parallel
  - スパコンの並列性を多数の（逐次）プログラムでも活用可能にする
  - ジョブ内キューを作って同時実行数を制限しながら順次実行



プログラムの並列化

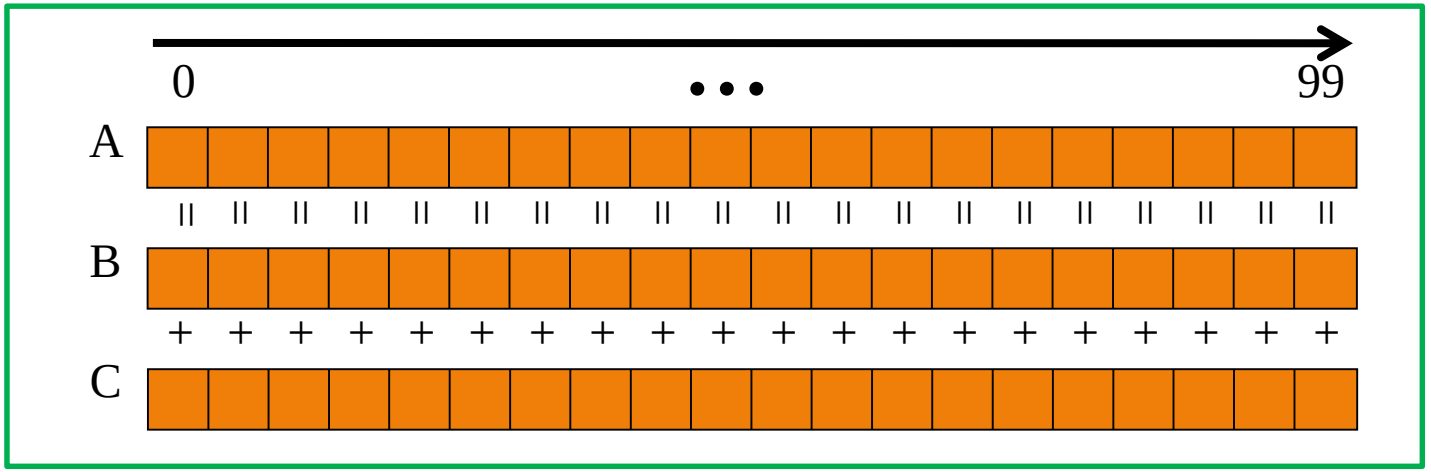
- 計算を分割して、計算ノードやコアや演算器に分担させる
  - 必要に応じて、データも分割
  - 必要に応じて、通信を呼び出し
- プログラムを「並列化」する必要がある場合：
  - 一つのプログラムで、一つの入力データに対して、出来るだけ速く計算したい
  - 一つの計算ノードではメモリが不足する



どのように分割するか？

並列化されていないプログラムの例：2つのベクトルの和を計算

- 0番目から99番目までの要素を順に計算

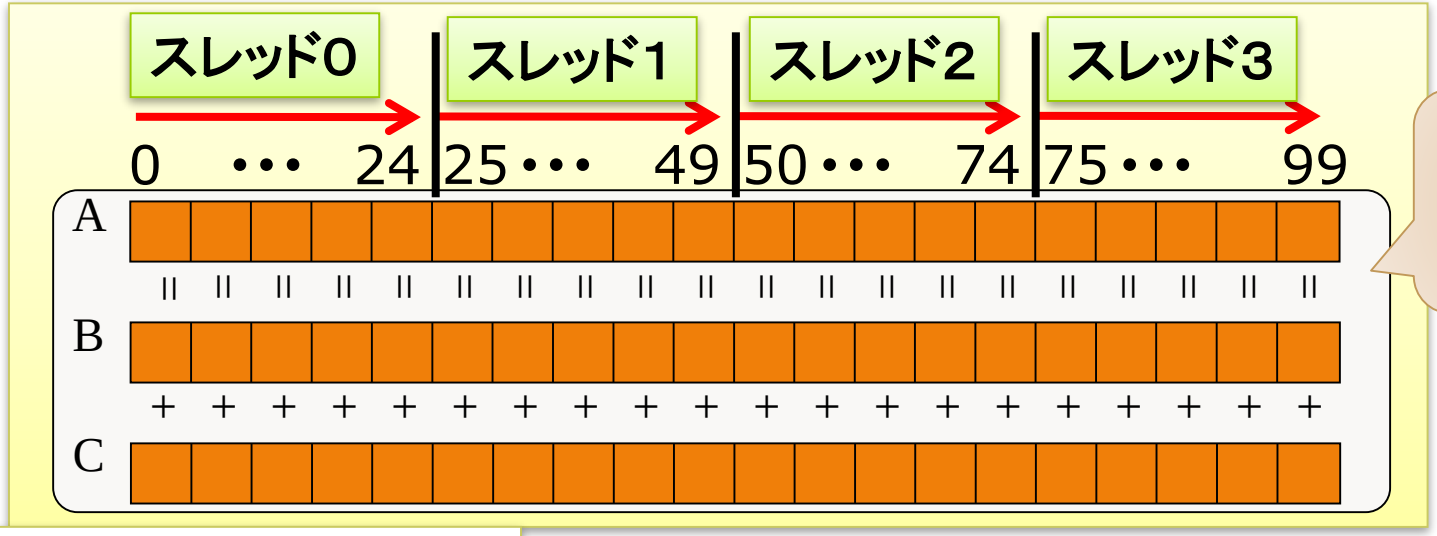


プログラム

```
double A[100], B[100], C[100];
   ...
for (i = 0; i < 100; i++)
    A[i] = B[i] + C[i];
```

計算だけを分割：スレッド並列

- スレッド ＝ 同じ記憶空間を共有しながら
  プログラムの実行を進める流れ




```
double A[100],B[100],C[100];
  ...
for (i=0; i<25; i
  A[i] = B[i] + C
```

スレッド0

```
double A[100],B[100],C[100];
  ...
for (i=25; i<50; i
  A[i] = B[i] + C[
```

スレッド1

```
double A[100],B[100],C[100];
  ...
for (i=50; i<75; i++
  A[i] = B[i] + C[i
```

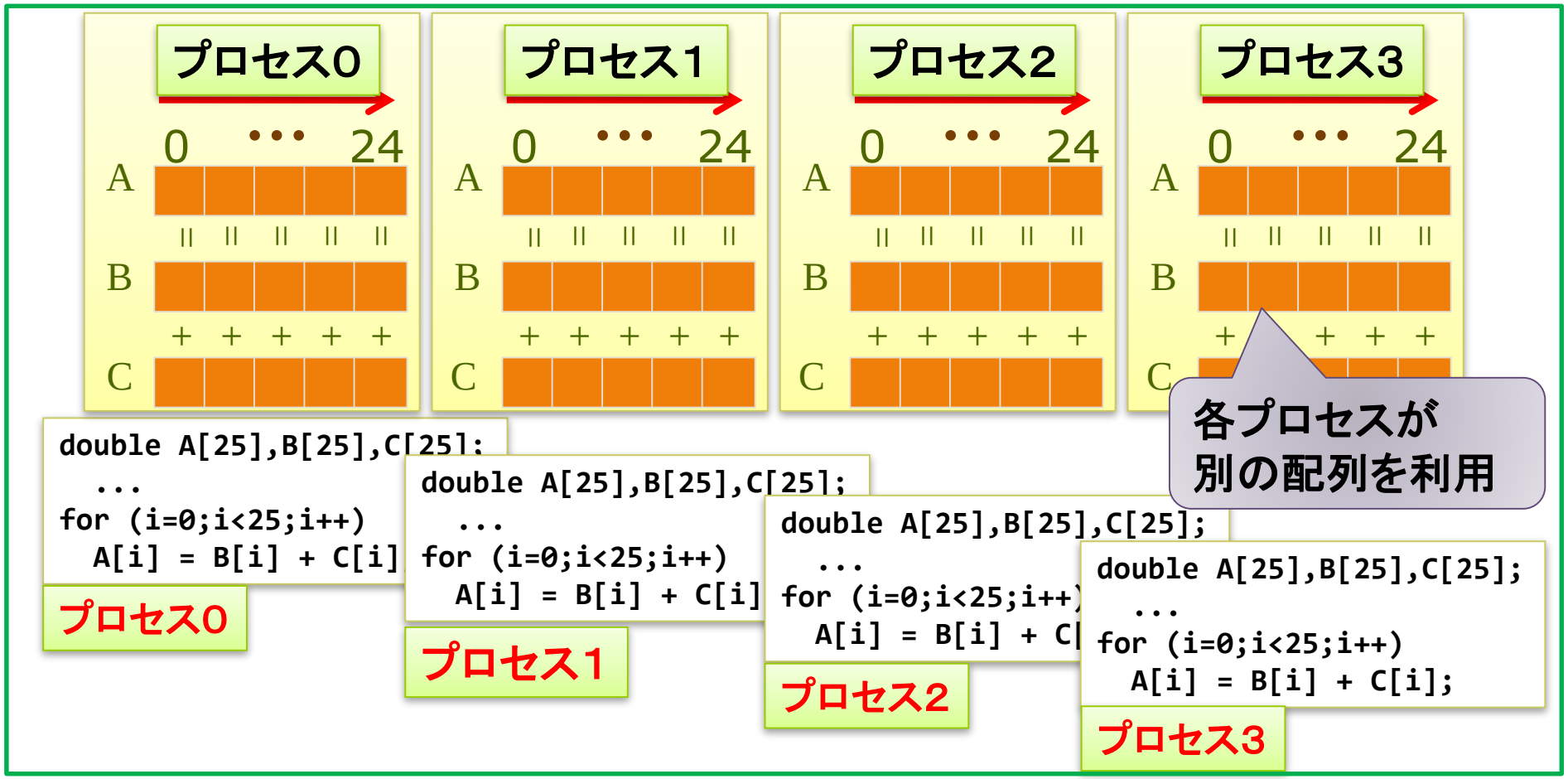
スレッド2

```
double A[100],B[100],C[100];
  ...
for (i=75; i<100; i++)
  A[i] = B[i] + C[i];
```

スレッド3

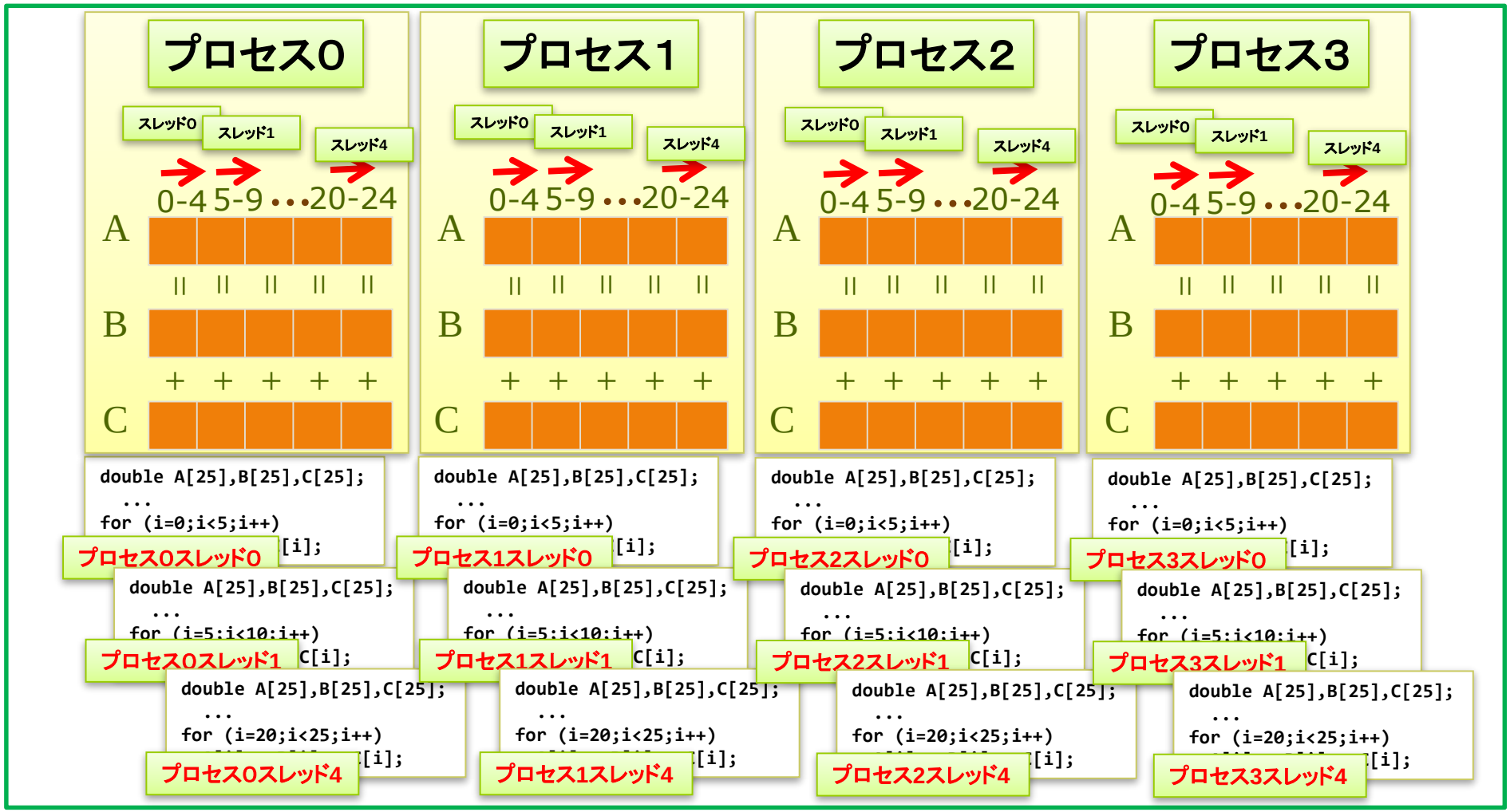
計算とデータを分割: プロセス並列

- プロセス ＝ それぞれ独立した記憶空間を持って
  プログラムの実行を進める流れ



ハイブリッド並列

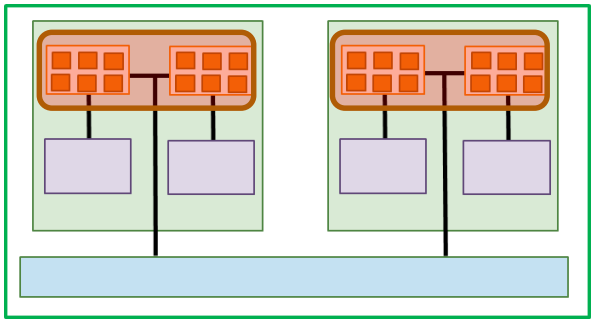
- 各プロセスの中で複数のスレッドを実行



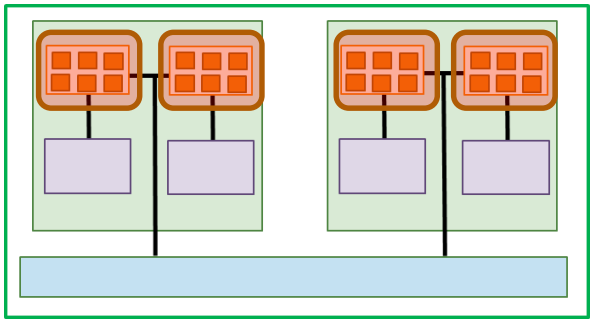
プロセス、スレッドとノード、コア

- プロセスは、一つのノードの中で実行
- スレッドは、プロセスに割り当てられた
  コアの一つで実行

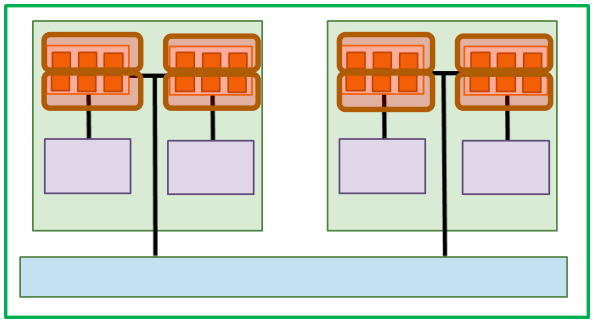
- 例）2ノード x 12コアでの計算
  - 一つのコアに一つのスレッドを割り当てる場合



2プロセス x 12スレッド



4プロセス x 6スレッド



8プロセス x 3スレッド

アクセラレータ、SIMD演算器での計算

- アクセラレータ
  - 専用のプログラム言語やインタフェースで、「アクセラレータに任せる計算」を記述
  - CUDA C（GPU用）の例

```
__global__ void vectorAdd
(const float *A, const float *B, float *C, int numElements)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if(i < numElements)
    C[i] = A[i] + B[i];
}
```

```
int main(void)
{
  ...
  vectorAdd<<<blocksPerGrid, threadsPerBlock>>>
    (d_A, d_B, d_C, numElements);
  ...
}
```

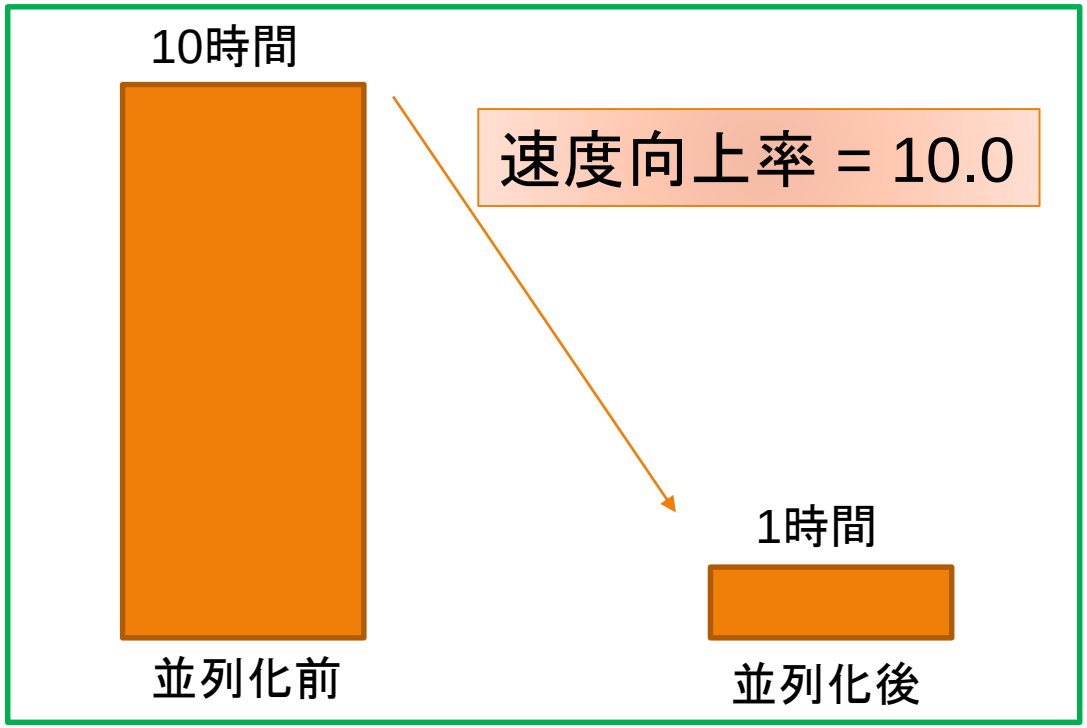
  - もっと簡単に記述できる方法も登場
    - OpenACC, OpenMP 4.0

- SIMD演算器
  - 内部では、「SIMD命令」で並列処理
  - 多くの場合、コンパイラが自動的にSIMD命令に変換

```
__m512 ax = _mm512_load_ps(&a[i]);
__m512 bx = _mm512_load_ps(&b[i]);
sumx = _mm512_fmadd_ps(ax, bx, sumx);
```

プログラム並列化の効果

- 速度向上率＝
  (並列化前の実行時間) / (並列化後の実行時間)
  - 並列化によって何倍速くなったか



並列化に対する期待と現実

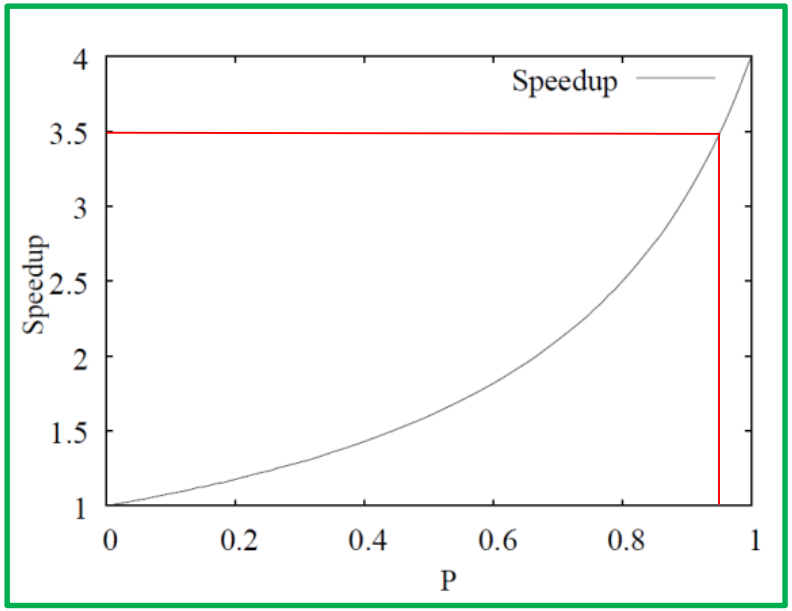
- 期待:
  「CPUを 4台使うんだから，並列化で 4倍速くなって欲しい」
- 現実:
  「CPU 4台で 3倍くらい速くなれば十分だろう」
  - 探索問題などでは、台数より速く結果が得られることはある
- 主な理由
  - アムダールの法則
  - 負荷のバランス
  - 通信のコスト

アムダールの法則

- プログラム中の高速化した部分しか高速化されない

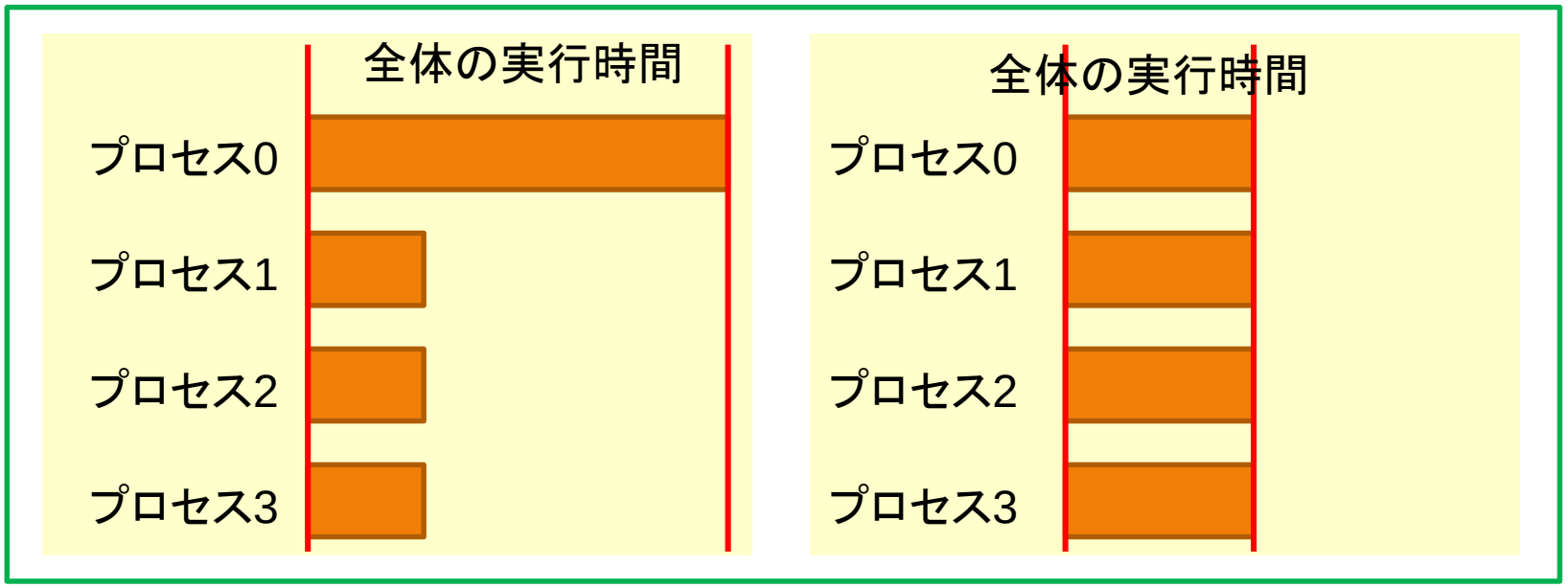
- 並列化にあてはめて考えると:
  並列化による性能向上率の理論的な限界
  ＝1/((1−P)+P/N)
  - P: プログラム中の並列化対象部分が全処理時間に占める割合
  - N: プロセス数

  - Example) N=4 で 3.5倍以上高速化するためには 95%以上の部分の並列化が必要



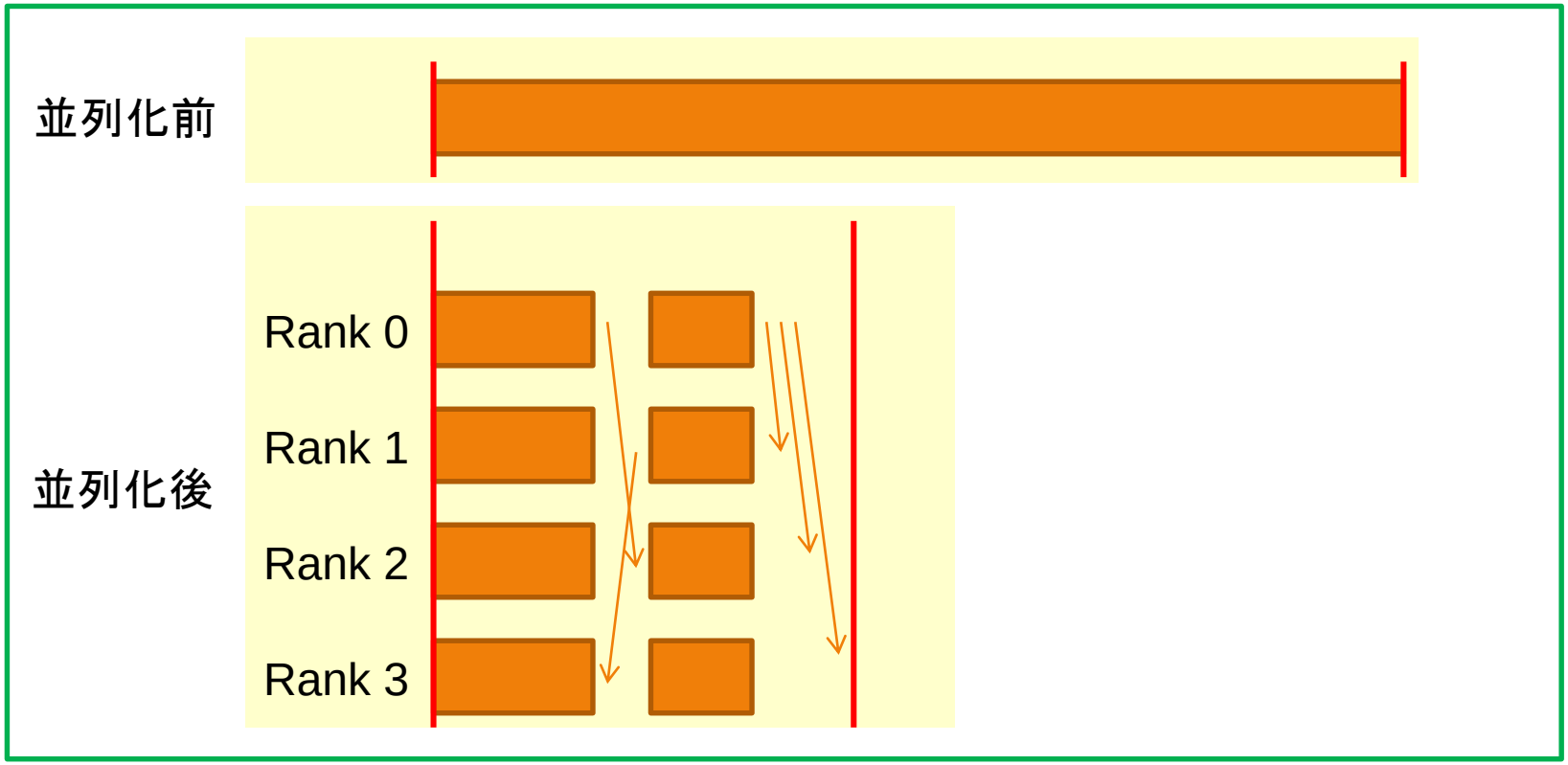
負荷のバランス

- 並列プログラムの実行時間は
  「最も遅いプロセスの実行時間」
  である



通信時間

- 並列化前は不要だった時間



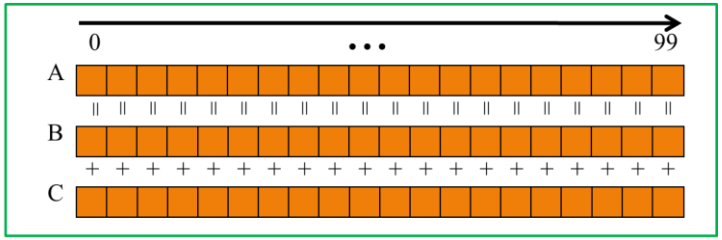
並列化以外の高速化も重要

- コンパイラの最適化オプションを試す
- キャッシュメモリやレジスタの利用効率を上げるようプログラムを改良する
- 無駄な計算を省く

それでも速度が遅ければ、プログラムを並列化

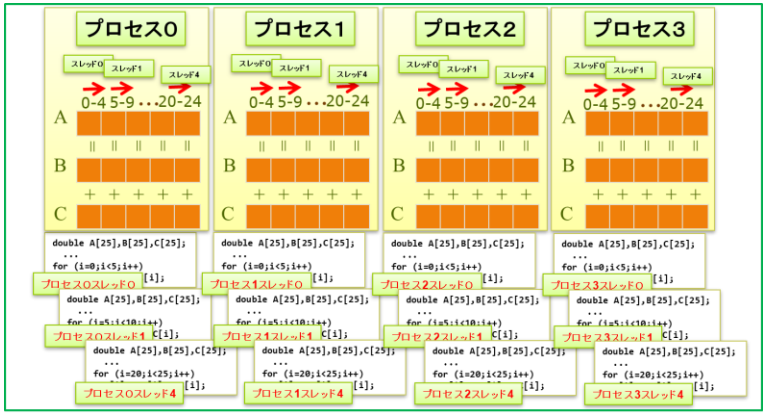
誰がプログラムを並列化するか？

1. 既に誰かが並列化したプログラムを使う
2. 誰かが並列化したライブラリを使う
3. コンパイラに並列化させる
4. 自分で並列化する



プログラム

```
double A[100], B[100], C[100];
  ...
for (i = 0; i < 100; i++)
    A[i] = B[i] + C[i];
```



1. 既に誰かが並列化したプログラム

- スーパーコンピュータセンターで用意されている
  アプリケーションを確認
  - 並列化されていれば、マニュアルに「プロセス数」や「スレッド数」の
    指定方法があるはず

- 並列化されたオープンソースソフトウェアの利用
  - コンパイル方法や実行方法の詳細は、センターの Webページ等を参照
  - 困ったらセンターに相談

2. 誰かが並列化したライブラリ

- 主に行列計算の関数を提供
  - ITOに用意されている数値計算ライブラリ
    - Fujitsu SSLII
      - 連立1次方程式の直接解法・反復解法、逆行列、固有値問題、フーリエ変換、擬似乱数など
    - Intel Math Kernel Library
      - BLAS、LAPACK、ScaLAPACK、BLACS、PBLAS、Sparse BLAS、疎行列演算関数（PARDISO含む)、フーリエ変換、偏微分方程式、非線形最適化ソルバ、データフィッティング関数、GMP（多倍長計算）関数、ベクトル化数学ライブラリ(VML)、統計関数（疑似乱数生成含む）
    - NAG Library
      - Numerical Algorithm Group社によって開発された数値計算ライブラリ
    - FFTW
      - 離散フーリエ変換
    - PETSc
      - 偏微分方程式によって記述された問題を並列計算機上で高速に処理するための数値計算ライブラリ

3. コンパイラによる並列化

- コンパイラが、プログラムのスレッド並列化や、SIMD命令の挿入を自動的に適用

- ほとんどの C, C++, Fortranコンパイラで利用可能
  - GNU, Intel, Fujitsu, PGI, etc.

- 簡単なプログラムでは、それなりの効果
  - ループの中に関数呼び出しや条件分岐が入っていない
  - ループの繰り返し数が、コンパイル時に分かっている
  - ループの繰り返し順序が変わっても、計算結果が変わらない

  等

それでも速度が遅ければ、自分でプログラムを並列化

4. 自分で並列化

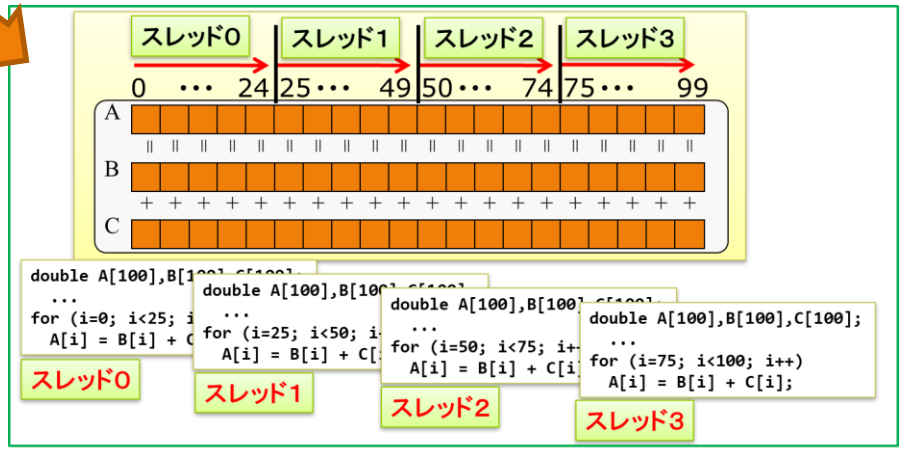
- 最も一般的な方法:
  - OpenMP でスレッド並列化
  - MPI (Message Passing Interface) でプロセス並列化
- 他にも:
  - Chapel
  - CAF (Co-array Fortran)
  - XcalableMP

OpenMPによる並列化

- プログラム中に「指示行」を追加
  - コンパイラが指示に従ってスレッド並列プログラムを作成

```
#include <omp.h>
double A[100], B[100], C[100];
   ...
#pragma omp parallel for
for (i = 0; i < 100; i++)
    A[i] = B[i] + C[i];
```

並列化指示行



MPI (Message Passing Interface)

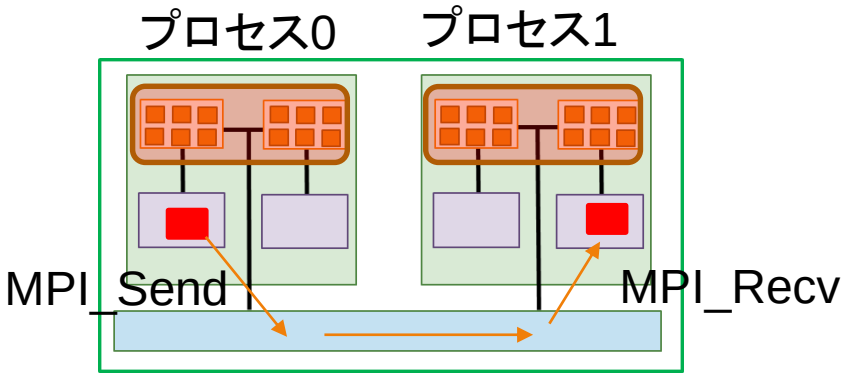
- 並列プログラム用に用意された通信関数群の定義
  - 例） プロセス0からプロセス1にデータを転送

```
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    ...
if (myid == 0)
  MPI_Send(&(a[5]), 1, MPI_DOUBLE, 1,
           0,  MPI_COMM_WORLD);
if (myid == 1)
  MPI_Recv(&(a[3]), 1, MPI_DOUBLE, 0,
           0, MPI_COMM_WORLD, &status);
```

自分のプロセス番号を取得

プロセス0がプロセス1に送信

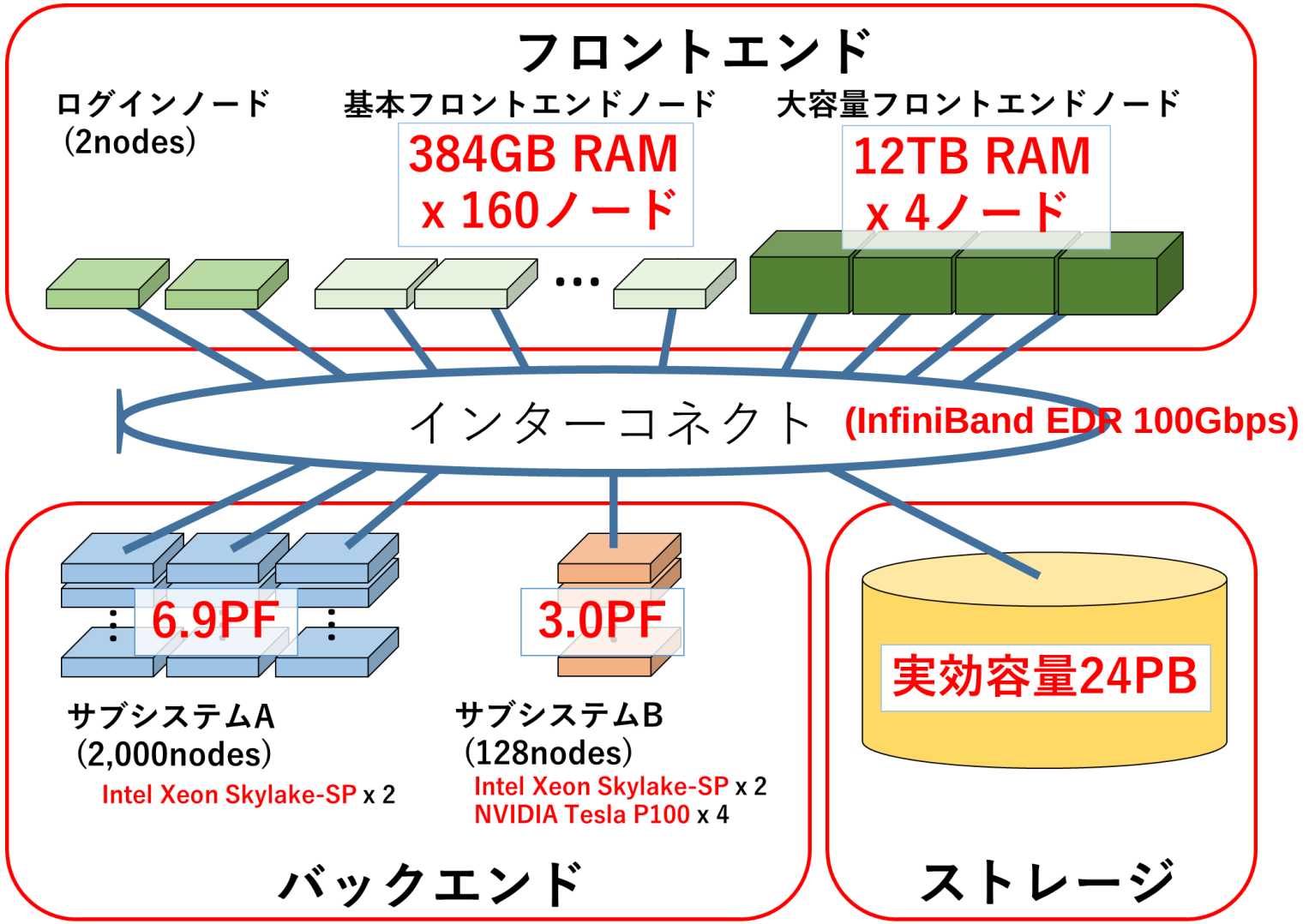
プロセス1がプロセス0から受信



今日の内容

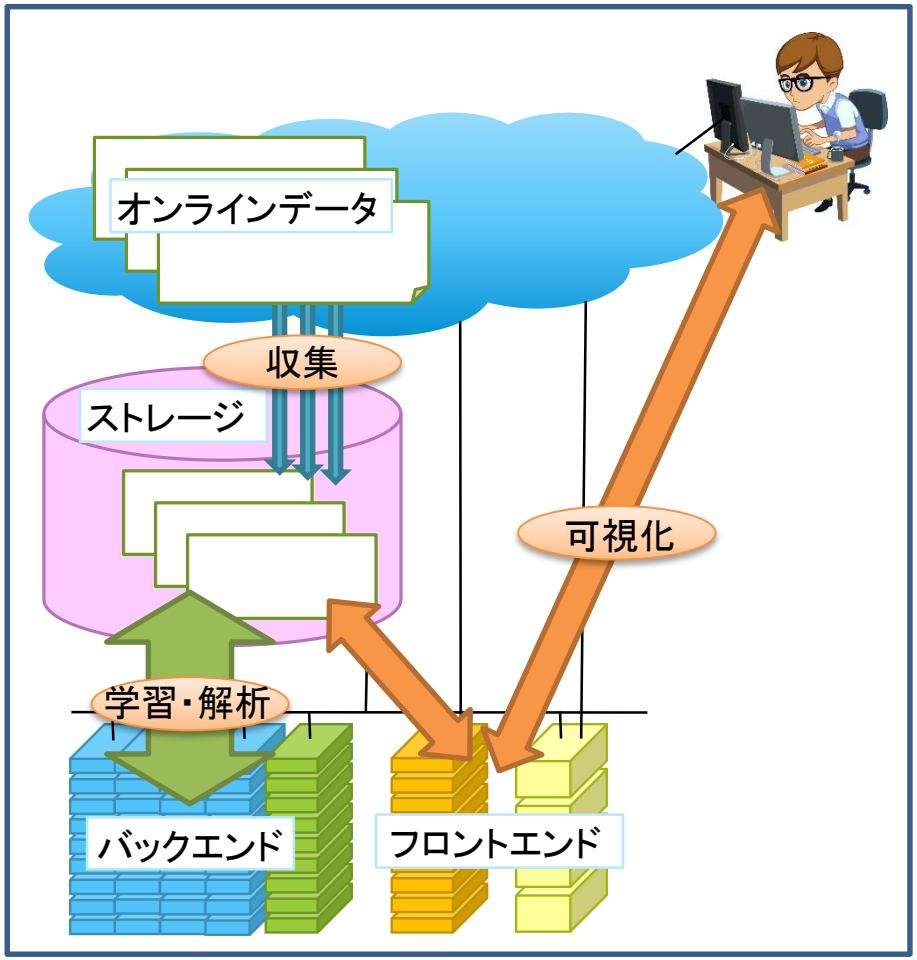
1. スーパーコンピュータの仕組み
2. スーパーコンピュータの使い方
3. 九州大学のスーパーコンピュータシステムITO紹介
4. スーパーコンピュータの開発競争と将来の展望
5. Q & A

九州大学のスーパーコンピュータ ITO



想定している利用例

- 大規模オンラインデータの
  収集、
  学習・解析、
  可視化
  の支援
  - 大容量ストレージ
  - 高速バックエンド
  - 大規模フロントエンド



ITOのソフトウェア

- コンパイラ、言語処理系
  - 富士通コンパイラ、Intelコンパイラ、PGIコンパイラ、CUDA、CUDA Fortran、OpenACC、Perl、Python
- 数値計算ライブラリ
  - SSL II、BLAS/LAPACK/ScaLAPACK、NAGライブラリ、FFTW、PETSc
- その他ライブラリ
  - HDF5、NetCDF、METIS
- 計算化学
  - Gaussian、Gaussview、CHARMM、VASP、Molpro、SCIGRESS、AMBER、GAMESS、GROMACS、LAMMPS、MODYLAS、NTChem、OpenMX、SALMON、SMASH、HΦ
- 流体・構造解析
  - Marc/Marc Mentat、MSC Nastran/Patran、ANSYS、OpenFOAM、FrontFlow/Red
- データ解析
  - SAS、ENVI/IDL、R
- 科学技術計算
  - Mathematica、MATLAB
- 機械学習
  - TensorFlow、Caffe、Chainer、CNTK
- 画像処理
  - FIELDVIEW、AVS

利用者からの追加要望にも対応
（可能な範囲で）

ITOの利用資格

- 大学、高等専門学校又は大学共同利用機関の教員及び学生
- 独立行政法人に所属する研究職員
- 学術研究を目的とする研究機関でセンター長が認めた機関に所属し、専ら研究に従事する者
- 外部資金を受けて学術研究を行う者
- 民間企業等に所属する者で、別に定める審査機関における審査を経て、センター長が認めた者
- その他特にセンター長が適当と認めた者

ITOの利用プラン （有料）

- 共有タイプ
  - 計算ノード群を複数のユーザで共有して利用
- ノード固定タイプ
  - 割り当てられた計算ノード群を準占有的に利用

プロジェクト利用 （原則無料）

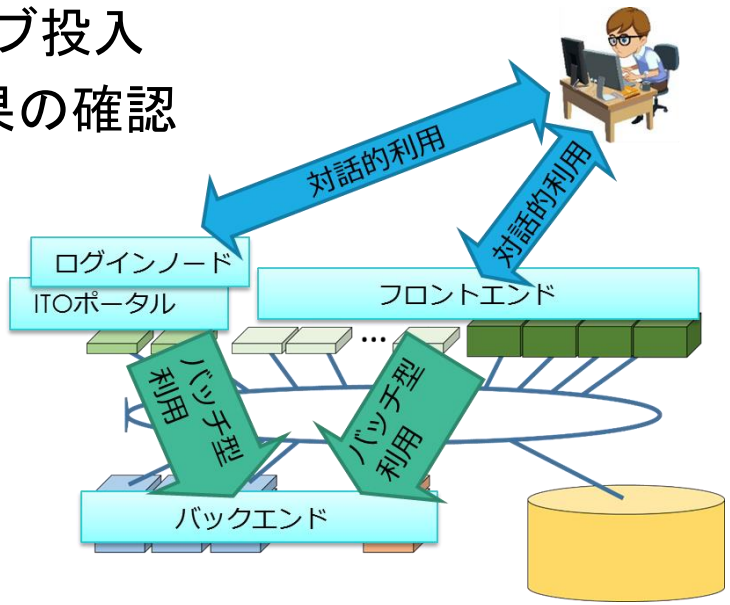
- 先端的計算科学研究プロジェクト
  - ITOを対象に九州大学が公募（毎年度末締切）
- HPCI (High Performance Computing Infrastructure)
  - 全国の共同利用スーパーコンピュータを対象に高度情報科学技術研究機構が公募
- JHPCN (学際大規模情報基盤共同利用・共同研究拠点)
  - 全国の共同利用スーパーコンピュータを対象にJHPCNが公募（毎年1月締切）
- JHPCN-Q
  - ITOを対象に、萌芽研究支援を目的として九州大学が公募（随時）

ITOの利用負担金

- サブシステム A
  - 4ノード（共有）: 2,960円 / 月
  - 4ノード（固定）: 23,600円 / 月
- サブシステム B
  - 1ノード（共有）: 2,100円 / 月
  - 1ノード（固定）: 17,000円 / 月
- 基本フロントエンド
  - 2CPU x 24時間まで同時予約可能: 900円 / 月
- 大規模フロントエンド
  - 8CPU x 24時間まで同時予約可能: 5,200円 / 月
- ストレージ
  - 10TB: 340円 / 月

申請から利用開始までの流れ

1. 利用申請
   - 利用プランの検討
   - 計算機利用申請書を記入
   - 申請書送付
   - 利用承認書発行（1週間程度）
2. 利用準備
   - 初期パスワード変更
   - 公開鍵登録
   - 必要なファイルのアップロード、もしくは作成
   - プログラムのコンパイル
3. 利用（フロントエンド）
   - 予約
   - ログインしてプログラム実行
4. 利用（バックエンド）
   - ジョブスクリプトの作成
   - ジョブ投入
   - 結果の確認



1. 利用申請
利用プランの検討

利用負担金表:
https://www.cc.kyushu-u.ac.jp/scp/service/fee_list/

- 共有タイプ or ノード固定タイプ
  - 最初は共有タイプの利用を推奨
- 利用システム
  - 対話的な利用が不要な場合
    - サブシステム A: CPUのみ、メモリ 192GB / ノード
    - サブシステム B: CPU + GPU利用、メモリ 384GB / ノード
  - 対話的な利用が必要な場合
    - 基本フロントエンド: 384GBメモリ / ノード
    - 大容量フロントエンド: 12TBメモリ / ノード
  - 上記を組み合わせて利用可能
- 使用ノード数
  - プログラムやデータによる
    - 使用メモリ量、一回の計算にかかる時間、同時に何個の計算を実行させるか、等
- ストレージ
  - 10TB or 100TB

例)今まで 4コア 16GBの PCで
1回1時間かかったプログラムを
1000個のデータについて実行したい

- 方法 1 プログラムを変更しない場合
・ITO サブシステムAを 4ノード利用
・4ノードで最大 16個のジョブ(データ)を同時実行(ノード当たり 4個ずつ)
・所要時間: 約 64時間

- 方法 2 GPUで 10倍高速化されたプログラムを利用できる場合
・ITO サブシステムBを 2ノード利用
・2ノードで最大 2個のジョブを同時実行
・所要時間: 約 50時間

- さらに、大量の計算結果からグラフを作成するために基本フロントエンド Sプラン
(2CPU * 24時間まで同時予約可能)

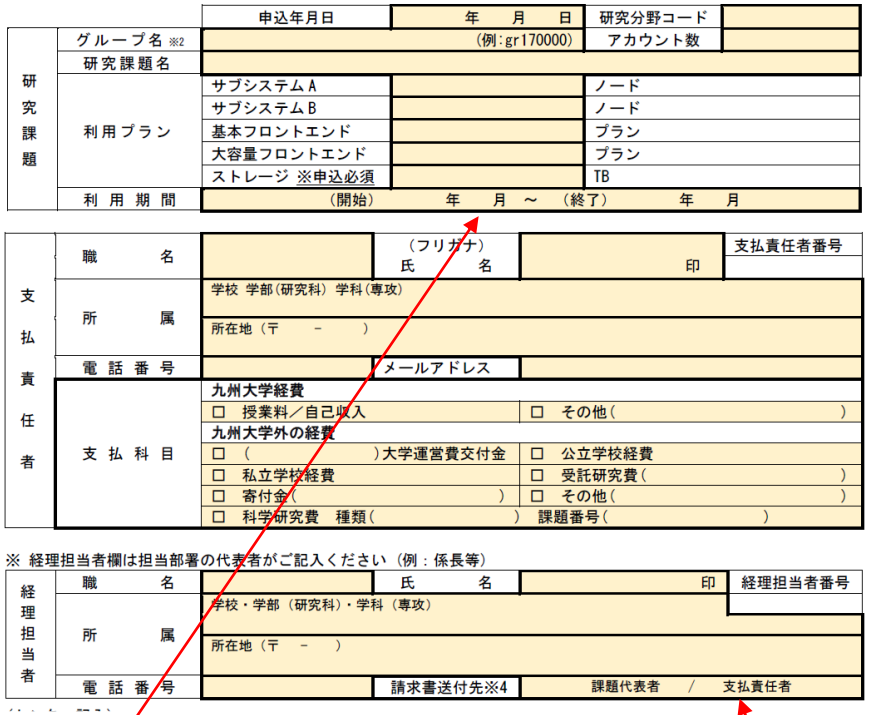
- ストレージは、とりあえず 10TB
(足りなくなったら、研究室に退避するか、容量を追加申請)

1. 利用申請
利用申請書の記入

利用申請書:
https://www.cc.kyushu-u.ac.jp/scp/service/guidance/application/

- 共有利用の申請書

1ページ目



研究課題		申込年月日	年 月 日	研究分野コード	
	グループ名 ※2	(例:gr170000)		アカウント数	
	研究課題名				
	利用プラン	サブシステムA		ノード	
		サブシステムB		ノード	
		基本フロントエンド		プラン	
		大容量フロントエンド		プラン	
		ストレージ ※申込必須		TB	
	利用期間	(開始) 年 月 ～ (終了) 年 月			

支払責任者	職名		(フリガナ) 氏名	印	支払責任者番号
	所属	学校 学部(研究科) 学科(専攻)			
		所在地(〒 -)			
	電話番号		メールアドレス		
	支払科目	九州大学経費			
		□ 授業料/自己収入	□ その他( )		
		九州大学外の経費			
		□ ( )大学運営費交付金	□ 公立学校経費		
		□ 私立学校経費	□ 受託研究費( )		
		□ 寄付金( )	□ その他( )		
		□ 科学研究費 種類( )	課題番号( )		

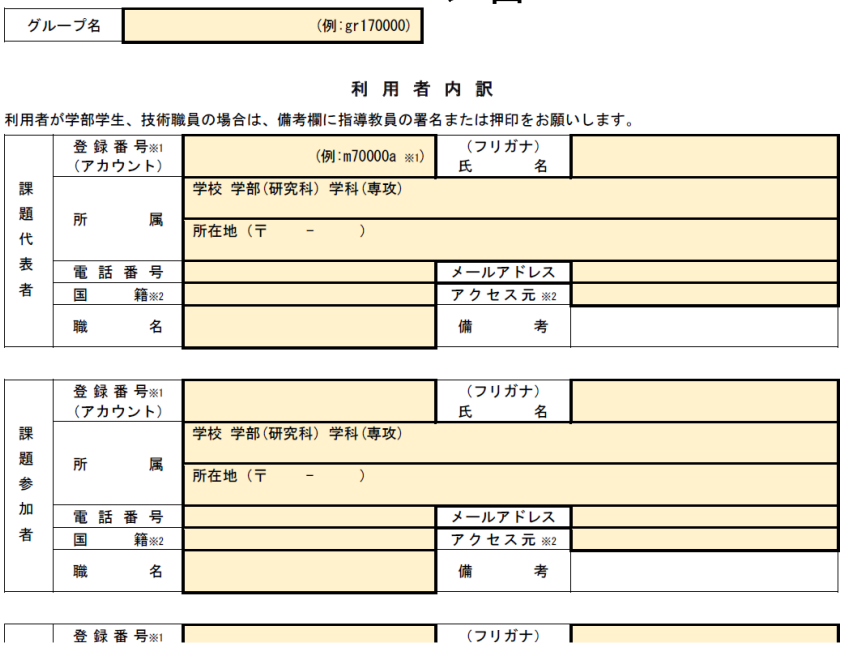
※ 経理担当者欄は担当部署の代表者がご記入ください (例:係長等)

経理担当者	職名		氏名	印	経理担当者番号
	所属	学校・学部(研究科)・学科(専攻)			
		所在地(〒 -)			
	電話番号		請求書送付先※4	課題代表者 / 支払責任者	

利用は月単位

研究費を管理している
会計担当の方に相談

2ページ目



グループ名	(例:gr170000)

利用者内訳

利用者が学部学生、技術職員の場合は、備考欄に指導教員の署名または押印をお願いします。

課題代表者	登録番号※1 (アカウント)	(例:m70000a ※1)	(フリガナ) 氏名	
	所属	学校 学部(研究科) 学科(専攻)		
		所在地(〒 -)		
	電話番号		メールアドレス	
	国籍※2		アクセス元 ※2	
	職名		備考	

課題参加者	登録番号※1 (アカウント)		(フリガナ) 氏名	
	所属	学校 学部(研究科) 学科(専攻)		
		所在地(〒 -)		
	電話番号		メールアドレス	
	国籍※2		アクセス元 ※2	
	職名		備考	

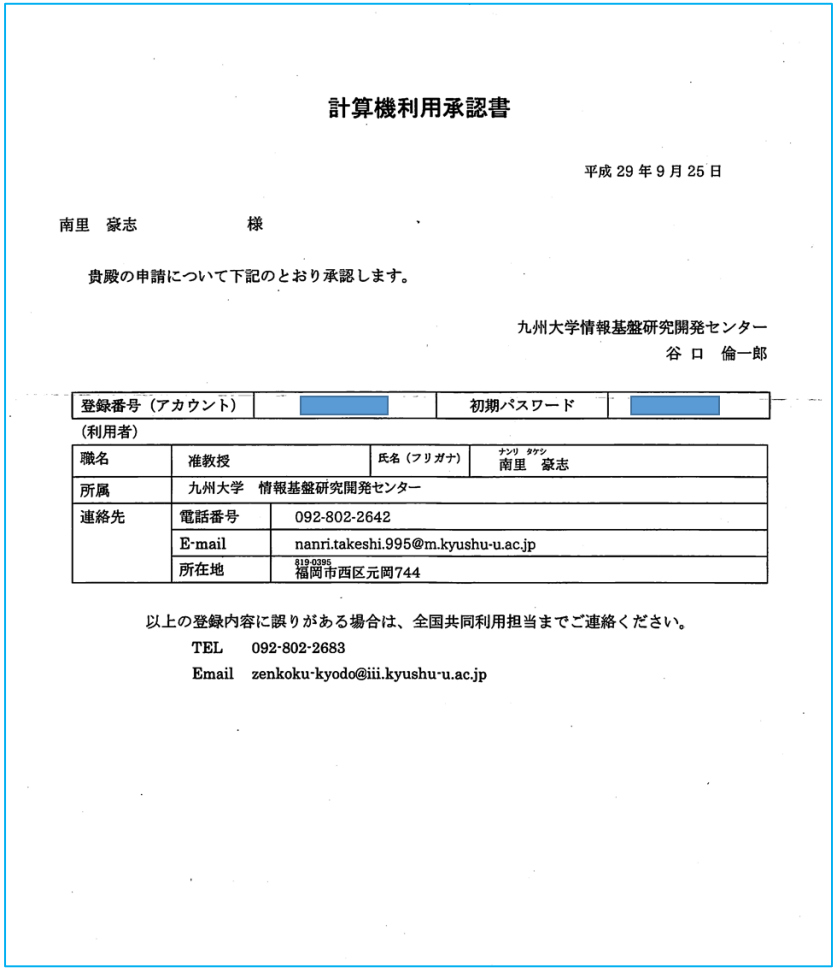
	登録番号※1		(フリガナ)

全利用者の情報
(外国籍の方の利用承認には
最大1ヶ月ほど要する)

1. 利用申請
利用承認書到着

- アカウント
- 初期パスワード

を確認



計算機利用承認書

平成 29 年 9 月 25 日

南里　豪志　　様

貴殿の申請について下記のとおり承認します。

九州大学情報基盤研究開発センター
谷 口　倫一郎

登録番号（アカウント）	[redacted]	初期パスワード	[redacted]

（利用者）

職名	准教授	氏名（フリガナ）	ナンリ タケシ 南里　豪志
所属	九州大学　情報基盤研究開発センター		
連絡先	電話番号	092-802-2642	
	E-mail	nanri.takeshi.995@m.kyushu-u.ac.jp	
	所在地	819-0395 福岡市西区元岡744	

以上の登録内容に誤りがある場合は、全国共同利用担当までご連絡ください。
TEL　092-802-2683
Email　zenkoku-kyodo@iii.kyushu-u.ac.jp

2. 利用準備
初期パスワード変更

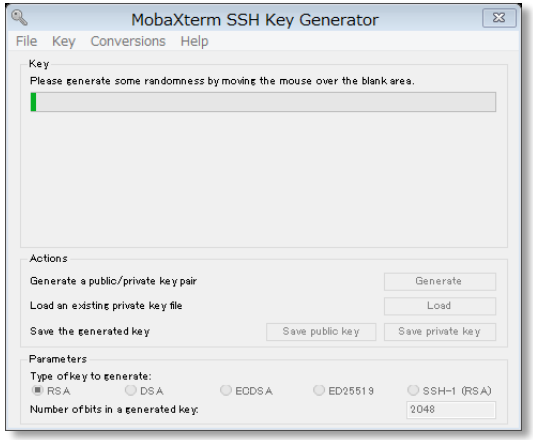
- ITOポータルにアクセス
  https://ito-portal.cc.kyushu-u.ac.jp/itoportal

- 配布されたユーザIDと初期パスワードでログイン
- 新しいパスワードを入力（2回）
  - 8文字以上
  - 変更後、反映されるまでに数分要します。

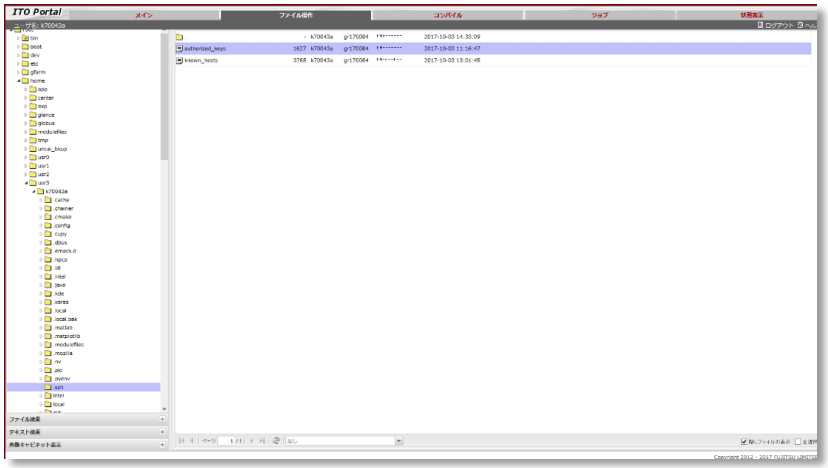
2. 利用準備
公開鍵登録

公開鍵登録:
https://www.cc.kyushu-u.ac.jp/scp/system/ITO/02_login/2.html

- 鍵ペア(秘密鍵、公開鍵)の用意
  - まだ作成していなければ、新規に作成
    - 注意: SSH-1(RSA)等の古い鍵ではログイン不可
      ⇒ 新規に鍵ペアを作成
    - Windows:
      端末エミュレータ(MobaXterm、Putty、TeraTerm等)に付属の作成ツール
    - macOS、Linux:
      ターミナルで ssh-keygen



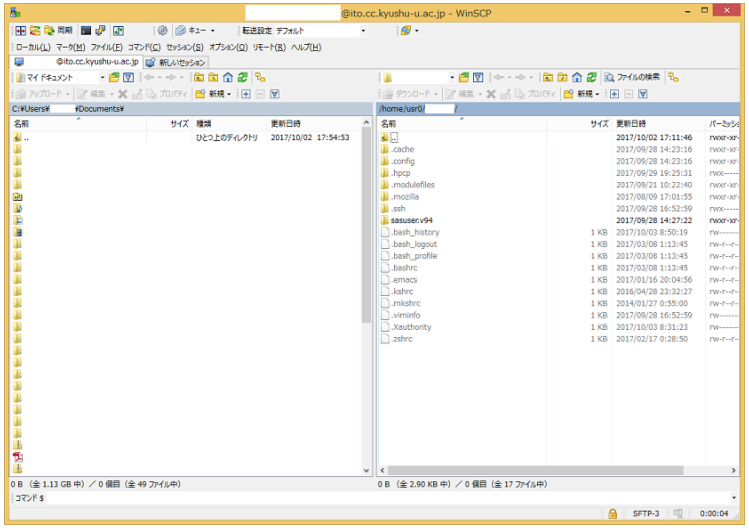
- 公開鍵の登録
  - 本センター Webサイトを参照



2. 利用準備
ファイルのアップロード

ファイル転送：
https://www.cc.kyushu-u.ac.jp/scp/system/ITO/02_login/

- Windowsの場合： WinSCP

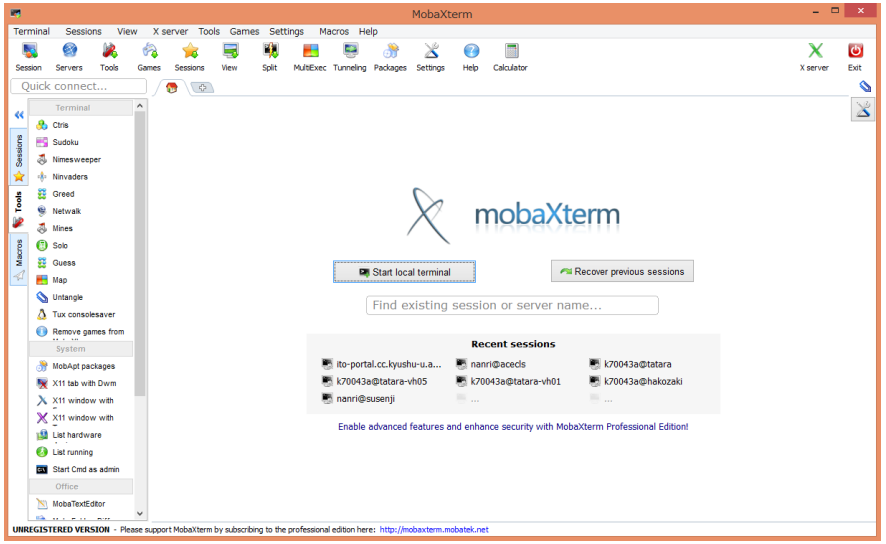


- macOS, Linuxの場合： scpコマンド

2. 利用準備
ログインノードにログイン

ファイル転送：
https://www.cc.kyushu-u.ac.jp/scp/system/ITO/02_login/

- Windowsの場合：
  - 端末エミュレータを利用
    - MobaXterm, TeraTerm, Putty, etc.
  - Windows上の Linux
    - Cygwin, Bash on Ubuntu on Windows, etc.



- macOS, Linuxの場合：
  - ターミナルから sshコマンドでログイン

```
$ ssh -i 秘密鍵ファイル名 -l ユーザ名 ito.cc.kyushu-u.ac.jp
```

2. 利用準備 プログラムのコンパイル

コンパイラの利用方法:
https://www.cc.kyushu-u.ac.jp/scp/software/

- 自作プログラムやオープンソースソフトウェアの場合

- 利用可能コンパイラ (C / C++ / Fortran)
  - Intel Compier
  - GNU
  - Fujitsu
  - PGI

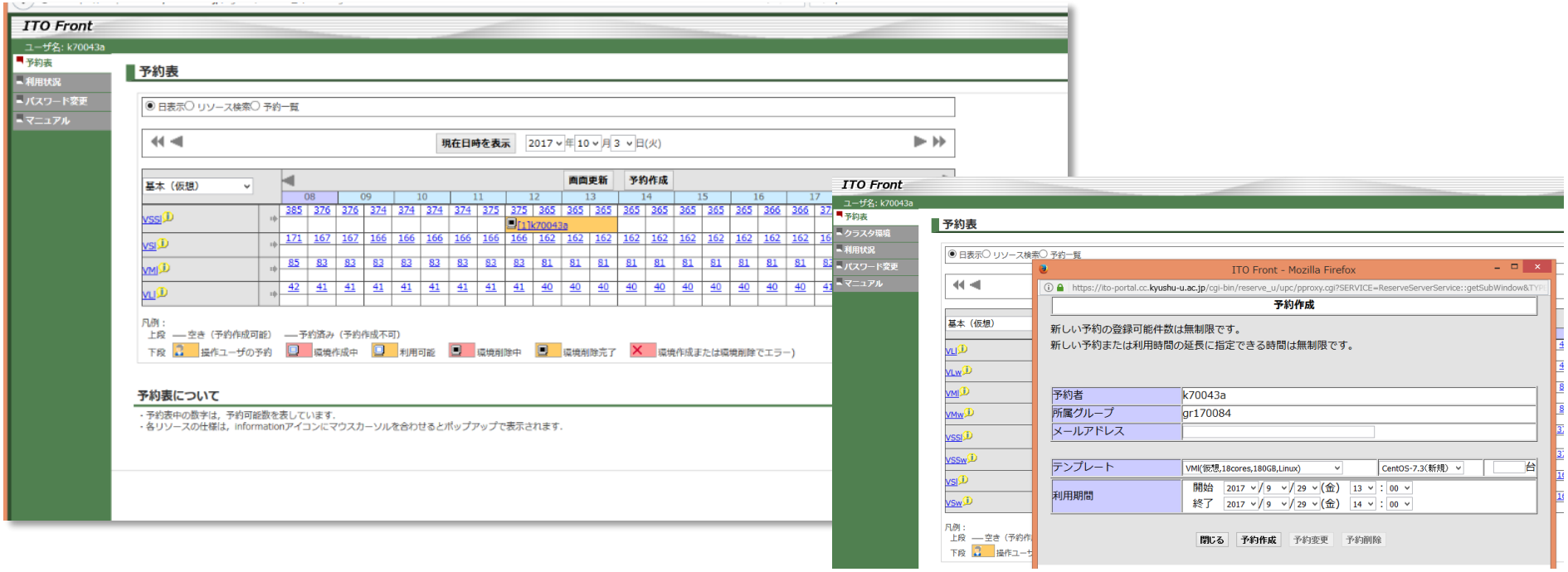
- 例）Intel Compiler 2018の場合

```
$ module load intel/2018.3
$ icc -ipo -O3 -no-prec-div -fp-model fast=2 -xHost test.c -o test
```

3. 利用(フロントエンド)
フロントエンドの予約

フロントエンドの予約:
https://www.cc.kyushu-u.ac.jp/scp/system/ITO/frontend/

1. 予約システムにログイン
   https://ito-portal.cc.kyushu-u.ac.jp/itofront
2. システムの空き状況確認
3. 自分の利用状況確認
4. 予約



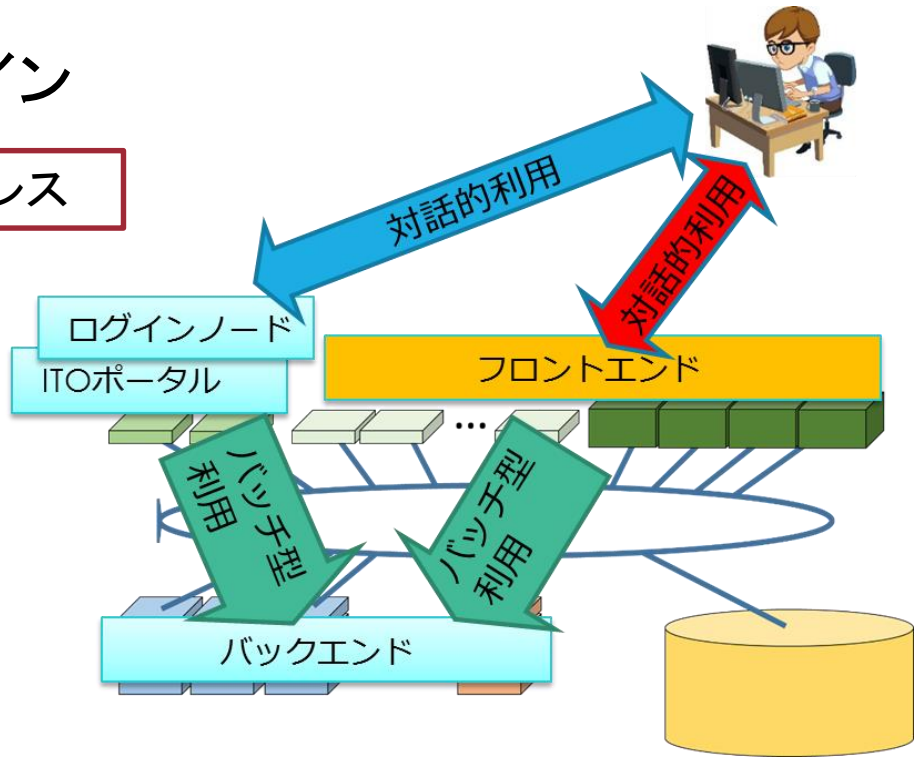
3. 利用(フロントエンド)
フロントエンドへのログイン

フロントエンドの予約:
https://www.cc.kyushu-u.ac.jp/scp/system/ITO/frontend/

- メールで利用ノード情報が通知
  - IPアドレスを確認

- ログインノードから、
  指定された IPアドレスにログイン

```
$ ssh -Y 予約したホストの IPアドレス
```



3. 利用(バックエンド) ジョブスクリプトの作成

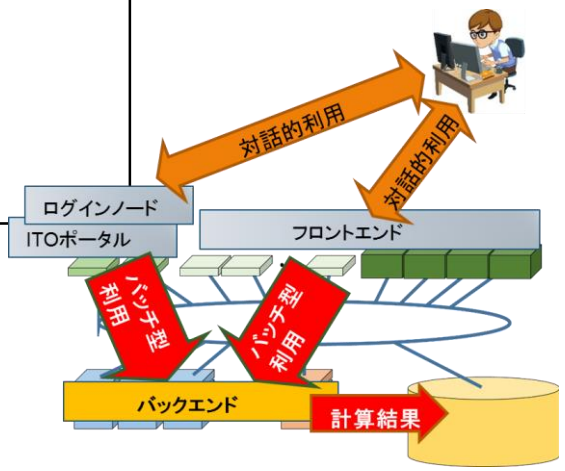
- テキストファイルとして作成
  - ログインノードで編集、もしくは PCで編集したものをアップロード
- 処理してほしい内容とジョブの内容を記述

- 例）サブシステム Aを 1ノード利用するプログラムの実行

```
#!/bin/sh

#PJM -L "vnode=1"
#PJM -L "vnode-core=36"
#PJM -L "rscunit=ito-a"
#PJM -L "rscgrp=ito-ss"
#PJM -L "elapse=02:00:00"

module load intel/2018.3
./test data1
```



3. 利用(バックエンド) ジョブ投入と結果確認

- ジョブの投入

```
$ pjsub test.sh
[INFO] PJM 0000 pjsub Job 28246 submitted.
```

- ジョブの状態確認

```
$ pjstat

ACCEPT QUEUED  STGIN  READY RUNING RUNOUT STGOUT   HOLD  ERROR   TOTAL
       0      0      0      0      1      0      0      0      0       1
s      0      0      0      0      1      0      0      0      0       1

JOB_ID     JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM NODE_REQUIRE    VNODE  CORE V_MEM
28246      test-mpifc NM RNE k70043a  10/04 09:34:06  0000:05:00 -                1      36   unlimited
```

- ジョブの結果確認

```
$ ls
test    test.c    test.sh    test.sh.i28246   test.sh.o28246
```

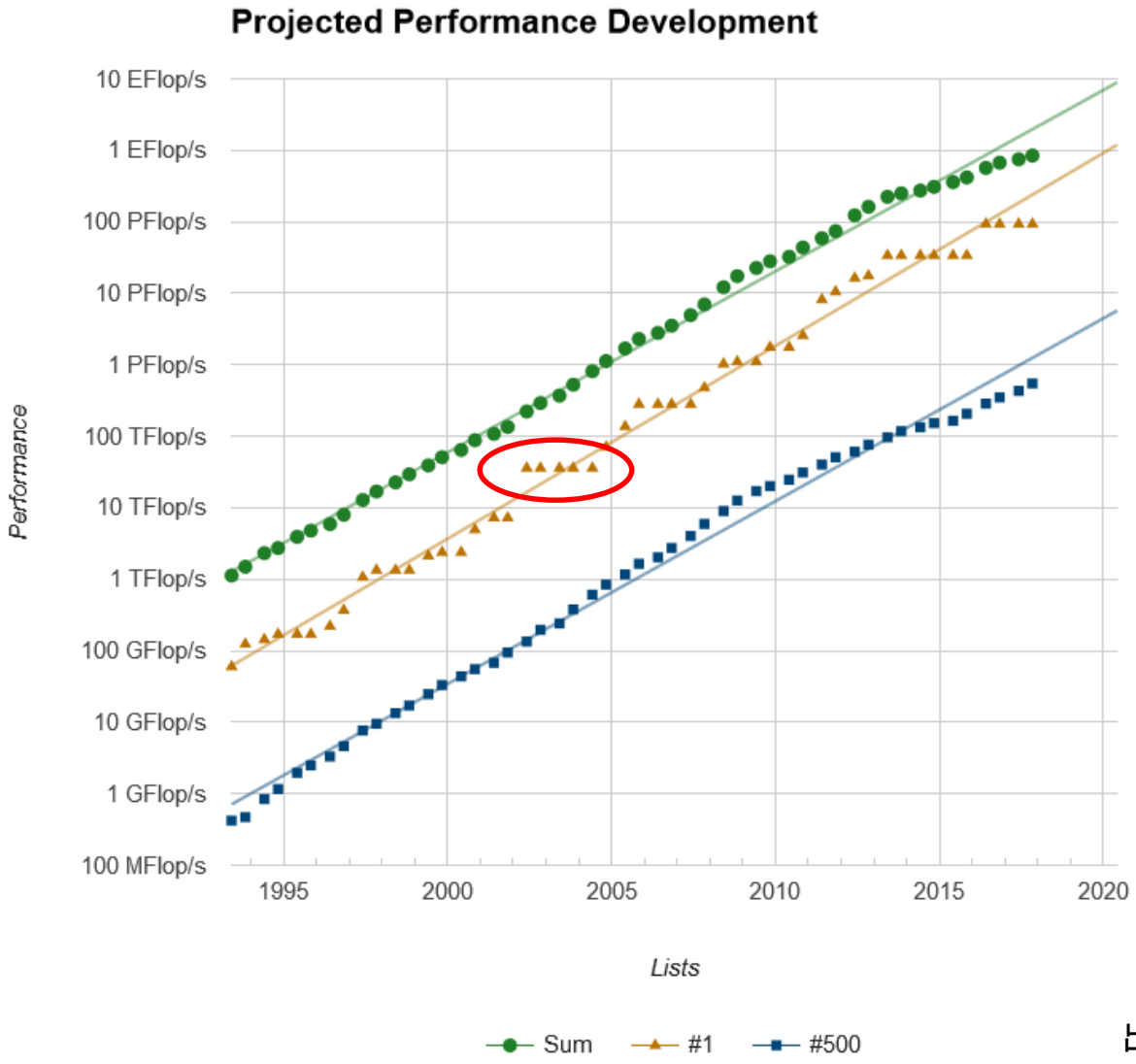
今日の内容

1. スーパーコンピュータの仕組み
2. スーパーコンピュータの使い方
3. 九州大学のスーパーコンピュータシステムITO紹介
4. スーパーコンピュータの開発競争と将来の展望
5. Q & A

「世界最速のスーパーコンピュータ」とは？

- 最も有名な指標：Top500（https://www.top500.org/）
  - スーパーコンピュータ性能比較リスト
  - 稼働中のスーパーコンピュータの1位～500位を掲載
  - 「LINPACKベンチマークプログラム」の性能で順位付け
    - （巨大な）連立一次方程式の解を求める計算
  - 第1回は1993年
  - 毎年6月と11月に更新

性能の推移

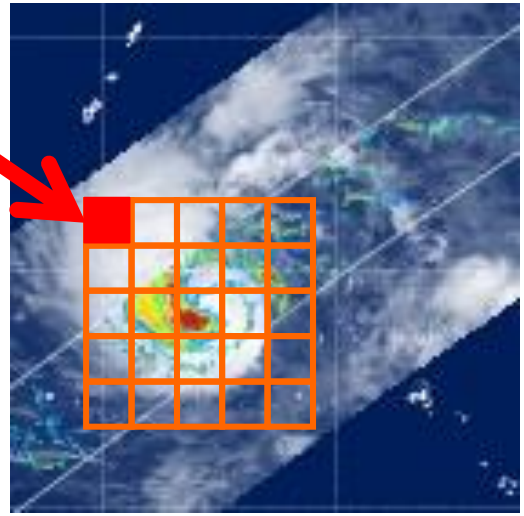


出典：https://www.top500.org/

過去の#1システムの例：地球シミュレータ

- 2002年3月稼働開始、主にNECが開発、地球シミュレータセンター（JAMSTEC）に設置

- 開発目標：10km四方（赤道近辺）の精度で地球全体の大気循環をシミュレートする
  - それまでは 100km四方
  - 台風の発生過程：100km四方だと台風が台風に見えない



- 地球シミュレータの成果
  - 台風の進路予測
    - 5.5km四方で地球全体をシミュレート（日本近辺は 2.78km四方）
    - 海底探査船「ちきゅう」に、高精度の台風進路予測を到達予定の3日前までに提供
  - 台風の発生予測
    - 過去10年間について、シミュレーションによる台風発生回数が実際の値とほぼ一致
  - CO2の増加に伴う温暖化の予測
    - 2040年には年間の真夏日日数が約20日増加，平均気温が約2度上昇.

Top500における地球シミュレータの性能

- Linpack 性能 35.8 TFLOPS＝1秒あたり35兆回の実数計算
  - 断トツの 1位
  - 2位から10位までの計算機の総演算性能を上回る（2002年6月時点）
  - 理論最大性能 41.0 TFLOPS、実行効率87%

- “Computenik”
  「計算機分野での Sputnik だ！」　(in New York Times)
  - by Jack　Dongarra教授（テネシー大学教授、Top500サイトの創始者）

- 地球シミュレータは2年半の間#1の座を守り続ける
- アメリカのスーパーコンピュータ開発に火を付けた

PFLOPSの実現、演算加速器の普及

- Blue Gene/L （IBM、LLNL、2004年11月～）
  - 2004.11　70.7 TFLOPS （理論最大性能 91.8 TFLOPS）
  - 2005.06　136.8 TFLOPS （理論最大性能 183.5 TFLOPS）
    - 世界で初めて 100 TFLOPS に到達
  - 2005.11　280.6 TFLOPS （理論最大性能 367.0 TFLOPS）
  - 2007.11　478.2 TFLOPS （理論最大性能 596.4 TFLOPS）
- Roadrunner （IBM、LANL、2008年6月～）
  - 1.026 PFLOPS （理論最大性能 1.3758 PFLOPS）
  - 世界で初めて 1 PFLOPS に到達
  - Opteron + PowerXCellによるヘテロジニアスシステム
  - この頃から通常のCPUよりもシンプルで高並列な演算加速器（アクセラレータ、特にGPU）の採用が増加
    - 電力効率増加、演算効率低下、汎用性と使いやすさはやや犠牲に
- その後、Tianhe（中国）、「京」（日本）などと10 PFLOPS級の戦いへ

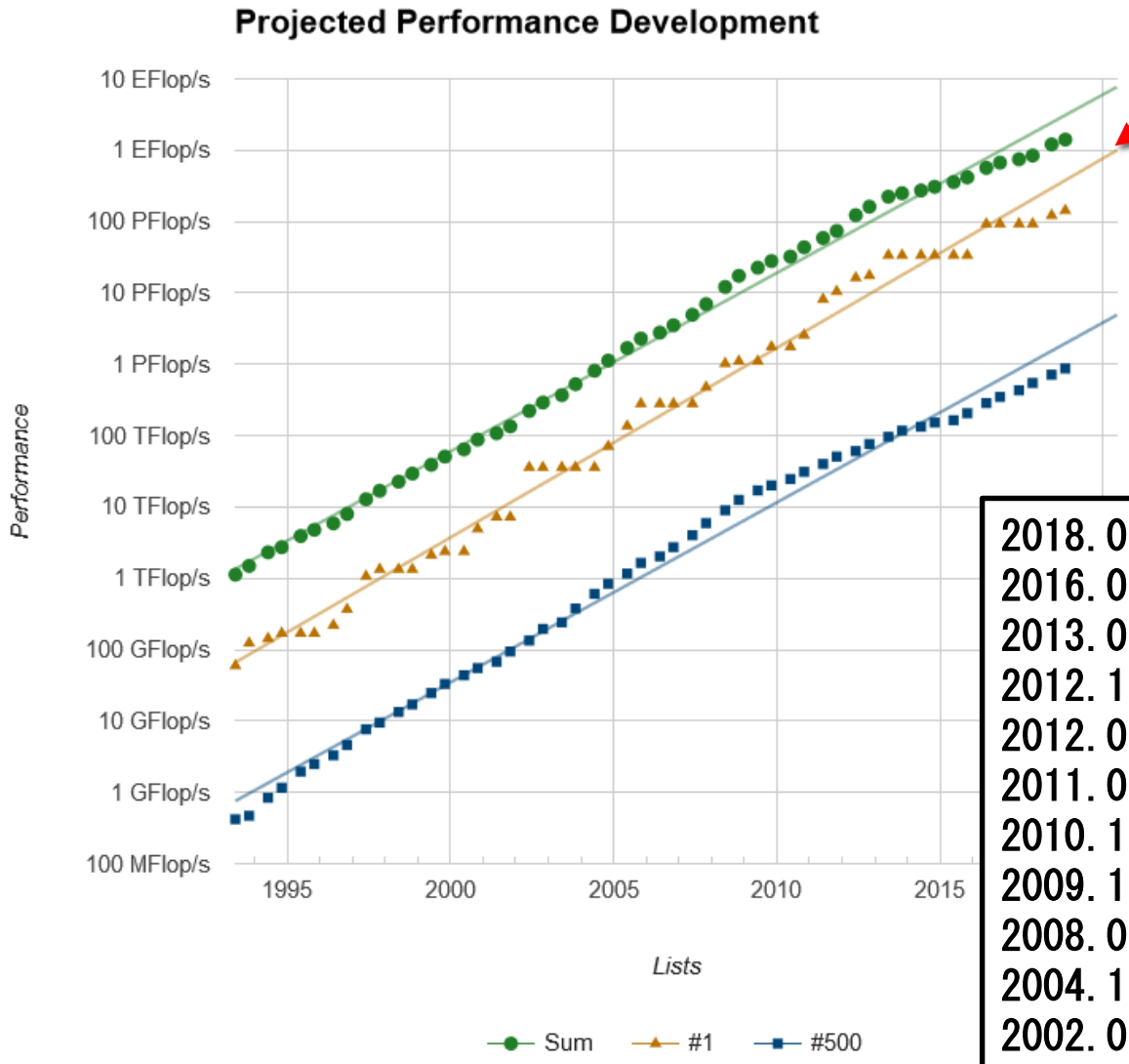
TOP500 2018年11月の状況

Rank	Name	Country	Linpack Perf (PFLOPS)	# nodes	Topology
1	Summit	USA	143.5	4608	Fat Tree
2	Sierra	USA	94.6	4320	Fat Tree
3	Sunway TaihuLight	China	93.0	40960	Fat Tree
4	Tianhe-2A	China	61.4	16000	Fat Tree
5	Piz Daint	Switzerland	21.2	5272	DragonFly
6	Trinity	USA	20.2	19420	DragonFly
7	AI Bridging Cloud Infrastructure (ABCI)	Japan	19.9	1088	Fat Tree
8	SuperMUC-NG	Germany	19.5	6480	Fat Tree
9	Titan	USA	17.6	18688	3D Torus
10	Sequoia	USA	17.2	98304	5D Torus

国別合計性能

Country	Total Perf (PFLOPS)	Share
USA	806	37.7%
China	757	31.0%
Japan	170	7.7%
Germany	86	4.3%

世界最速システムの変遷



2021年頃には
1 EFLOPSに到達する？

2018.06-	SUMMIT
2016.06-2017.11	Sunway TaihuLight
2013.06-2015.11	Tianhe-2
2012.11	Titan
2012.06	Sequoia
2011.06-2011.11	「京」コンピュータ
2010.11	Tianhe-1A
2009.11-2010.06	Jaguar
2008.06-2009.06	Roadrunner
2004.11-2007.11	BlueGene/L
2002.06-2004.06	地球シミュレータ
2000.11-2001.11	ASCI White

スーパーコンピュータのトレンド：どのようなシステムが上位にランクインしているのか？

- 2000年代前半まで
  - まとめて計算を行う仕組みを持った強力なCPUと高速なメモリを搭載した計算機を並べる
  - ベクトル型並列計算機、ベクトルプロセッサ
  - 多くの構成要素をスパコン専用に開発
- 2000年代後半以降
  - 1台の計算機の性能はやや控えめだが、ネットワークで大量に繋いで全体として高性能
  - 超並列計算機（Massively Parallel Processing）、マルチコアCPU
  - パソコン向けに近いパーツを多用（開発費削減、下方展開）
- 2010年代
  - さらに規模の大きなMPP、アクセラレータ（GPU）やメニーコアプロセッサの活用
  - 電力効率の良いシステムの追求

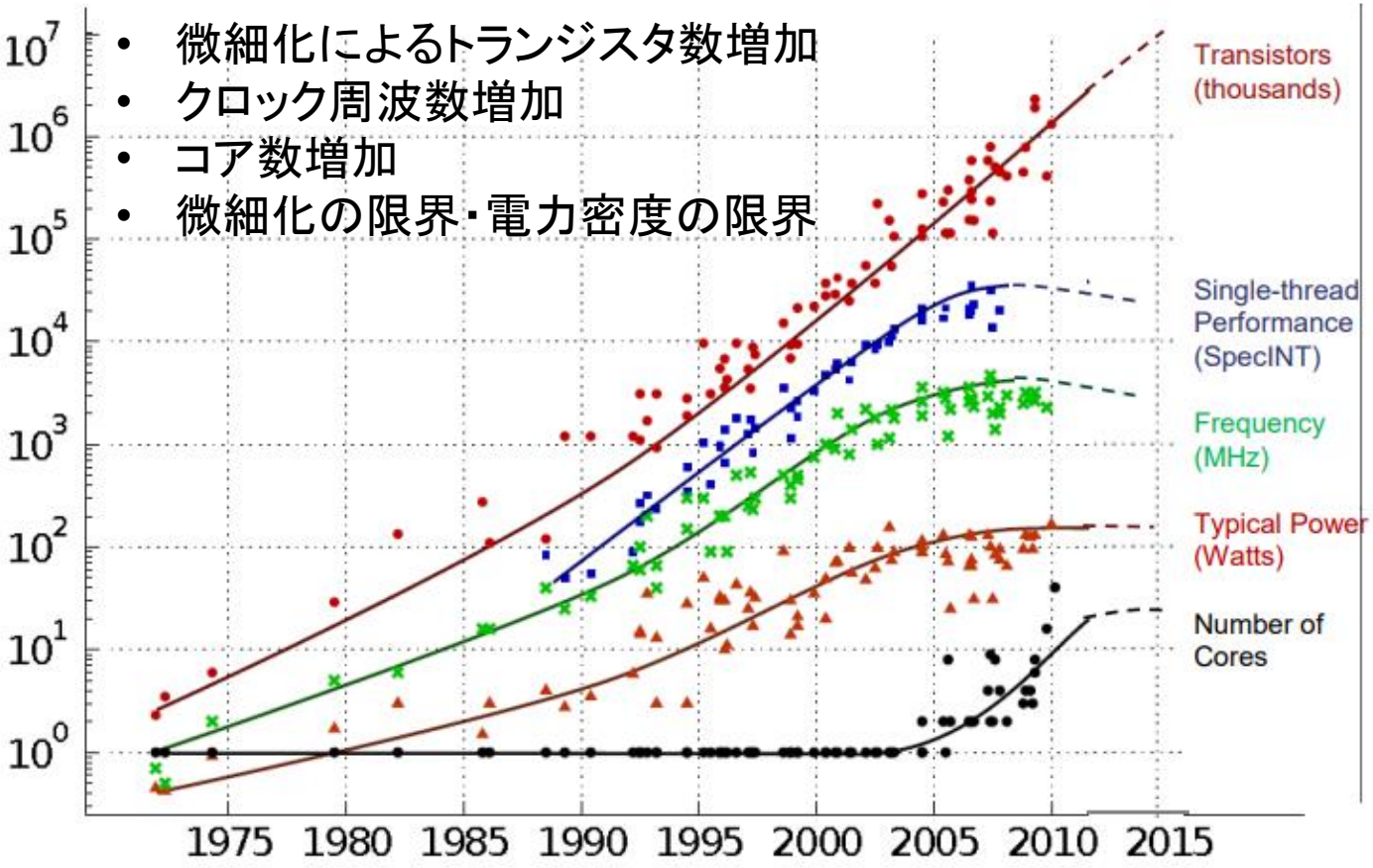
TOP500の課題、ベンチマークの多様化

- TOP500の抱える問題
  - 意味のある計算をしていない、実アプリで行う計算とかけ離れている、時間も電気代もかかる
  - 「連立一次方程式の求解が全てではない」「現在のスーパーコンピュータの性能を十分に反映していない」「現代のスーパーコンピュータランキングとして妥当ではない」
  - ⇔過去のシステムとの比較、安定性の確認、理想的な最大性能の確認

- TOP500以外のランキングへの注目も高まる
  - Green500 （http://www.green500.org）
    - TOP500の性能を消費電力で割った電力対性能比
  - HPCG （http://www.hpcg-benchmark.org）
    - 前処理付き共役勾配法アルゴリズムによる計算性能
  - Graph500 （http://www.graph500.org）
    - グラフ探索性能
  - IO500 （https://www.vi4io.org/io500/start）
    - ストレージ性能

スーパーコンピュータの性能向上を支えてきたもの

- 微細化によるトランジスタ数増加
- クロック周波数増加
- コア数増加
- 微細化の限界・電力密度の限界



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

引用元 http://www.lanl.gov/conferences/salishan/salishan2011/3moore.pdf

EFLOPSへの挑戦

- 1000 PFLOPS(1 EFLOPS)に向けて
  - 性能値の達成そのものは可能、とにかく大量に並べれば良い
  - 問題：お金・電力・設置空間が足りない、故障が増えて全体を安定稼働させ続けられない
  - 米国、中国、欧州、日本がそれぞれ計画を進めている
    - 膨大な開発費、半導体の微細化が難航、……達成は2022年頃か
  - アプリケーション性能は出るのか
    - 使いやすいHW/SWでなければ普及しない、利用者が困る
- 半導体の微細化が本当に終わってしまったあとはどうする？
  - 専用ハードウェアの活用、用途別の専用スーパーコンピュータ
  - 「従来のコンピュータ」とは異なるなにか
    - 例えば量子コンピュータ？

今日の内容

1. スーパーコンピュータの仕組み
2. スーパーコンピュータの使い方
3. 九州大学のスーパーコンピュータシステムITO紹介
4. スーパーコンピュータの開発競争と将来の展望
5. Q & A