科学ソフトウェアのウェブ・アプリケーション化 ～RIDGE パイプラインの場合～

江口 智士

福岡大学理学部物理科学科

2020 年 2 月 14 日

共同研究者: 柴垣 翔太、端山 和大、固武 慶 (福岡大学)

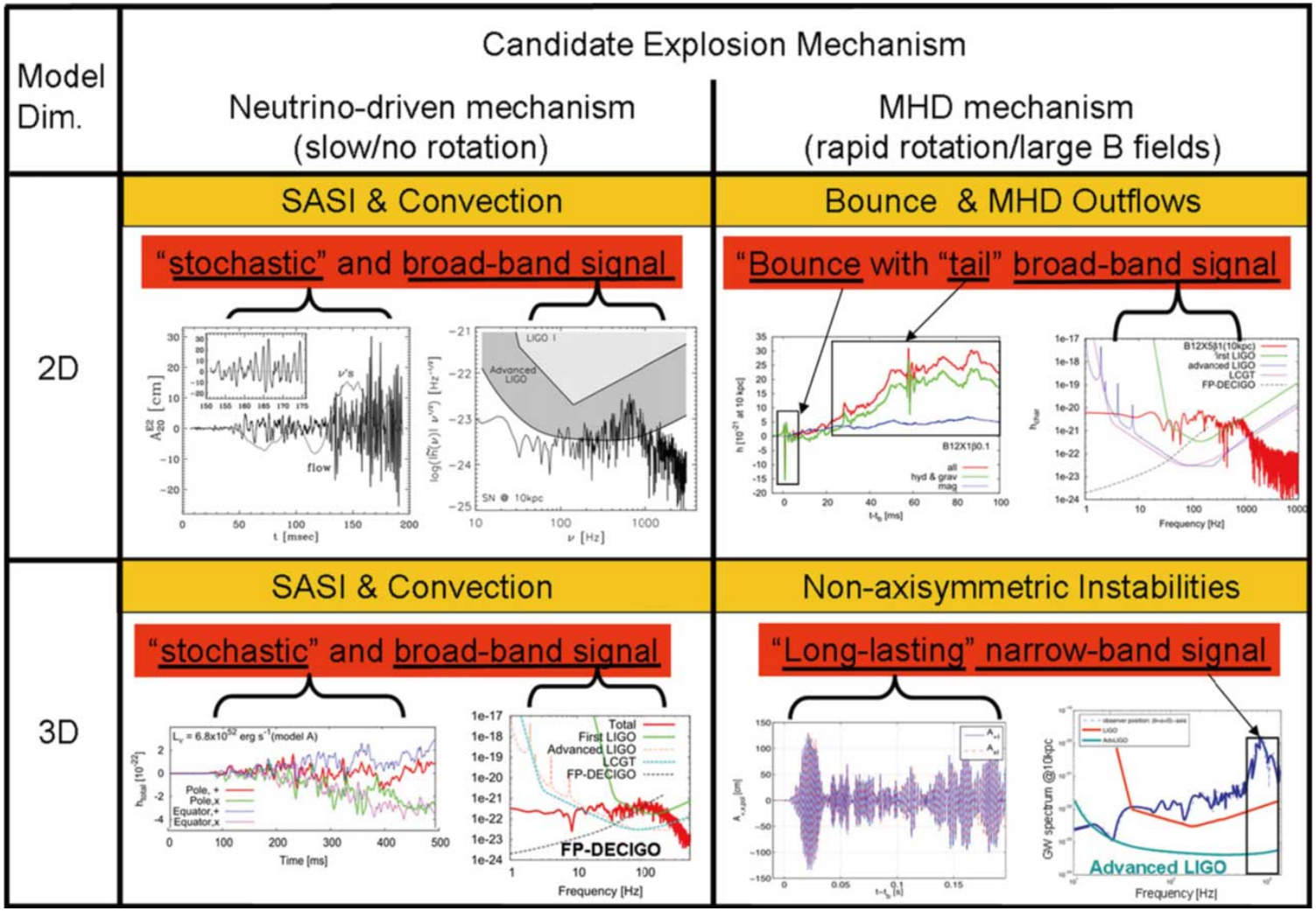
福岡大学 爆発天体研究所 (REISEP)

- 数値シミュレーション
- 電波〜ガンマ線 (電磁波) 観測
- ニュートリノ観測
- 重力波観測

を駆使して、重力崩壊型超新星爆発のメカニズムを解明する!

電磁波観測 + ニュートリノ観測 + 重力波観測
= マルチメッセンジャー天文学

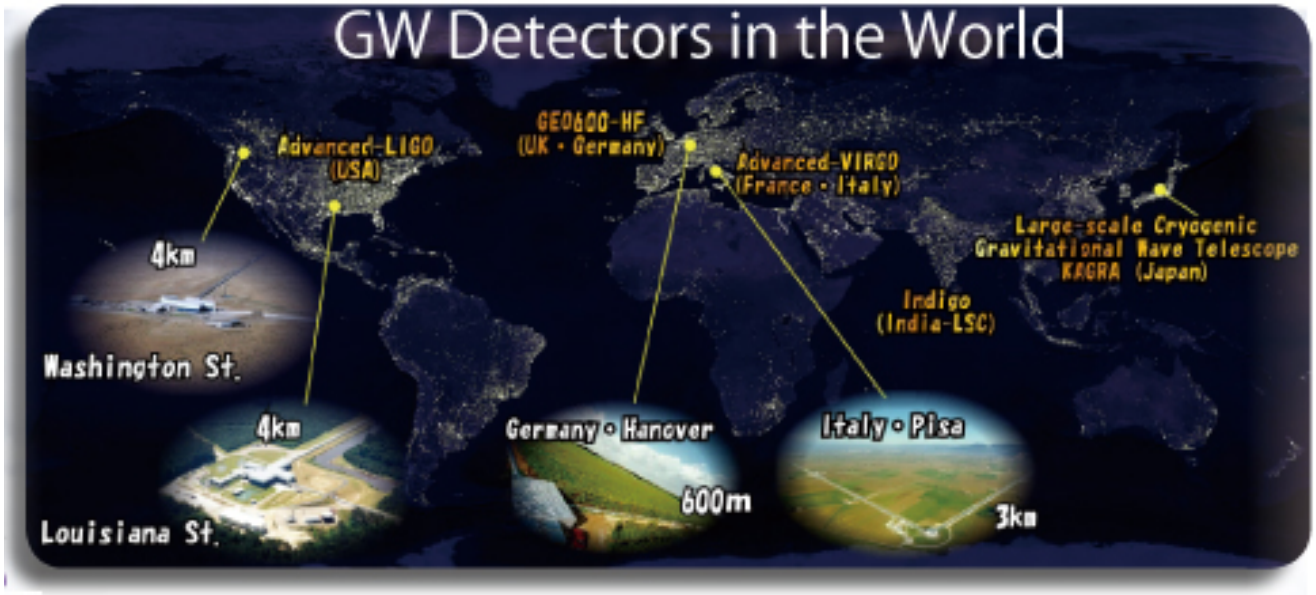
マルチメッセンジャー天文学と超新星爆発



Kotake K., 2013, CRPhy, 14, 318

RIDGE パイプライン

- コヒーレント・ネットワーク解析 (CNA)
  - 世界に散らばる重力波望遠鏡のデータをまとめて解析
  - 個々の検出器のノイズの影響を受けにくい
- RIDGE パイプライン：**Matlab** による CNA の実装の一つ
  (Hayama K., Mohanty S. D., Rakhmanov M., Desai S., ***2007***, CQGra, 24, S681)



http://www.icrr.u-tokyo.ac.jp/en/news/8190/

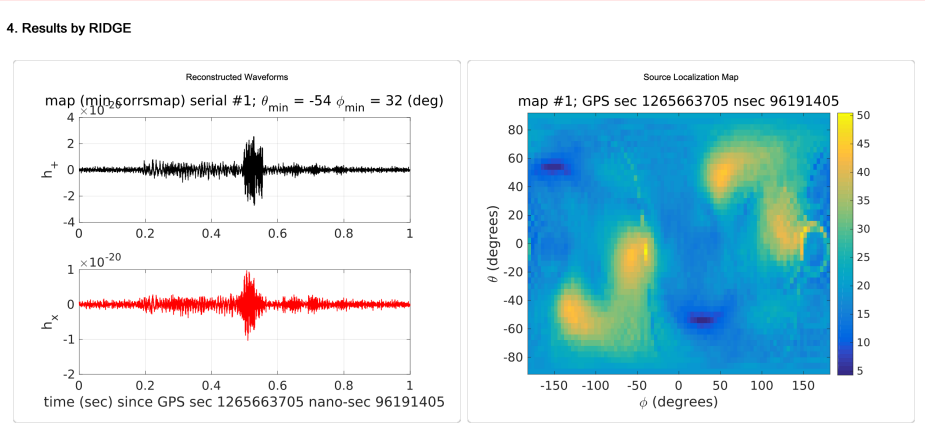
SNEGRAF

- **S**uper**N**ova **E**vent **Gra**vitational-wave-display in **F**ukuoka
- https://nibiru.sci.fukuoka-u.ac.jp/snegraf/
- ウェブ・ブラウザ経由で RIDGE パイプラインを利用可能
- 重力波の理論波形
  → 各重力波望遠鏡による検出のシミュレーション

入力：CSV ファイル

Time (sec)	h_+	$h_\times$
1.531696e-2	4.084765e-23	-5.649934e-23
1.537799e-2	3.092471e-23	-6.916069e-23
1.543903e-2	1.319925e-23	-8.146088e-23
1.550006e-2	-1.232760e-23	-9.425871e-23
1.556110e-2	-4.142535e-23	-1.080364e-22
⋮	⋮	⋮

出力：SVG ファイル



アーキテクチャの呪い (1)：初期設計時の懸念

32 ビット環境と 64 ビット環境で実行結果が微妙に変わる!

台形公式による $\int_0^1 \frac{4}{1+x^2} dx = 4 \arctan 1 = \pi$：定義部

```
#include <stdio.h>
inline static double f(const double x)
{
  return 4.0 / (1.0 + x * x);
}
static double integrate(const double a, const double b, const unsigned int n)
{
  const double dx = (b - a) / (double)n;
  double s = (f(a) + f(b)) / 2.0;
  double x = a;
  for (unsigned int i = 1; i < n; ++i) {
    x += dx;
    s += f(x);
  }
  return s * dx;
}
```

アーキテクチャの呪い (1)：初期設計時の懸念

32 ビット環境と 64 ビット環境で実行結果が微妙に変わる!

台形公式による $\int_0^1 \frac{4}{1+x^2} dx = 4 \arctan 1 = \pi$：定義部

```
#include <stdio.h>
inline static double f(const double x)
{
  return 4.0 / (1.0 + x * x);
}
static double integrate(const double a, const double b, const unsigned int n)
{
  const double dx = (b - a) / (double)n;
  double s = (f(a) + f(b)) / 2.0;
  double x = a;
  for (unsigned int i = 1; i < n; ++i) {
    x += dx;
    s += f(x);
  }
  return s * dx;
}
```

アーキテクチャの呪い (1)：初期設計時の懸念

台形公式による $\int_0^1 \frac{4}{1+x^2} dx = 4 \arctan 1 = \pi$：実行部

```
int main(int argc, char **argv)
{
  static volatile unsigned int N = 1U « 20; // N = 2^20 = 1048576
  const double pi = integrate(0.0, 1.0, N);
  printf("pi = %.16f\n", pi);
  return 0;
}
```

検証環境

- Ubuntu 19.10 amd64
- GCC 9.2.1

アーキテクチャの呪い (1) : 実行結果

64 ビット・バイナリ

- コンパイル : `gcc -g -O2 -Wall`
- 出力 : `pi = 3.14159265358966`**`61`**

32 ビット・バイナリ

- コンパイル : `gcc -g -O2 -Wall -m32`
- 出力 : `pi = 3.14159265358964`**`17`**

(「精度ギリギリの演算はするな!」という指摘はあると思いますが ...)

- なぜ?
- 気持ち悪くないですか?

アーキテクチャの呪い (1) : 実行結果

64 ビット・バイナリ

- コンパイル : `gcc -g -O2 -Wall`
- 出力 : `pi = 3.14159265358966`**`61`**

32 ビット・バイナリ

- コンパイル : `gcc -g -O2 -Wall -m32`
- 出力 : `pi = 3.14159265358964`**`17`**

(「精度ギリギリの演算はするな!」という指摘はあると思いますが ...)

- なぜ?
- 気持ち悪くないですか?

アーキテクチャの呪い (1) : 実行結果

64 ビット・バイナリ

- コンパイル : `gcc -g -O2 -Wall`
- 出力 : `pi = 3.14159265358966`**`61`**

32 ビット・バイナリ

- コンパイル : `gcc -g -O2 -Wall -m32`
- 出力 : `pi = 3.14159265358964`**`17`**

(「精度ギリギリの演算はするな!」という指摘はあると思いますが ...)

- なぜ?
- 気持ち悪くないですか?

アーキテクチャの呪い (2)：逆アセンブル

64 ビット・バイナリ

```
      for (unsigned int i = 1; i < n;
++i) {
10b8: 83 c0 01      add  $0x1,%eax
        return 4.0 / (1.0 + x * x);
10bb: 66 0f 28 d1   movapd %xmm1,%xmm2
10bf: f2 0f 59 d1   mulsd  %xmm1,%xmm2
10c3: f2 0f 58 d4   addsd  %xmm4,%xmm2
10c7: f2 0f 5e f2   divsd  %xmm2,%xmm6
        s += f(x);
10cb: f2 0f 58 c6   addsd  %xmm6,%xmm0
```

赤字：SSE2 命令

※自動並列化にはなっていない

32 ビット・バイナリ

```
      for (unsigned int i = 1; i < n;
++i) {
10e2: 83 c0 01      add    $0x1,%eax
        return 4.0 / (1.0 + x * x);
10e5: d9 c0         fld    %st(0)
10e7: d8 c9         fmul   %st(1),%st
10e9: d8 c3         fadd   %st(3),%st
10eb: d8 bb 40 e0 ff ff   fdivrs
-0x1fc0(%ebx)
        s += f(x);
10f1: de c2         faddp  %st,%st(2)
```

青字：x87 命令

- 同じ double 型の演算でも命令が異なる
- デフォルトのコンパイル・オプションが違うため

アーキテクチャの呪い (2)：逆アセンブル

64 ビット・バイナリ

```
      for (unsigned int i = 1; i < n;
++i) {
10b8: 83 c0 01      add  $0x1,%eax
        return 4.0 / (1.0 + x * x);
10bb: 66 0f 28 d1   movapd %xmm1,%xmm2
10bf: f2 0f 59 d1   mulsd  %xmm1,%xmm2
10c3: f2 0f 58 d4   addsd  %xmm4,%xmm2
10c7: f2 0f 5e f2   divsd  %xmm2,%xmm6
        s += f(x);
10cb: f2 0f 58 c6   addsd  %xmm6,%xmm0
```

32 ビット・バイナリ

```
      for (unsigned int i = 1; i < n;
++i) {
10e2: 83 c0 01      add    $0x1,%eax
        return 4.0 / (1.0 + x * x);
10e5: d9 c0         fld    %st(0)
10e7: d8 c9         fmul   %st(1),%st
10e9: d8 c3         fadd   %st(3),%st
10eb: d8 bb 40 e0 ff ff   fdivrs
-0x1fc0(%ebx)
        s += f(x);
10f1: de c2         faddp  %st,%st(2)
```

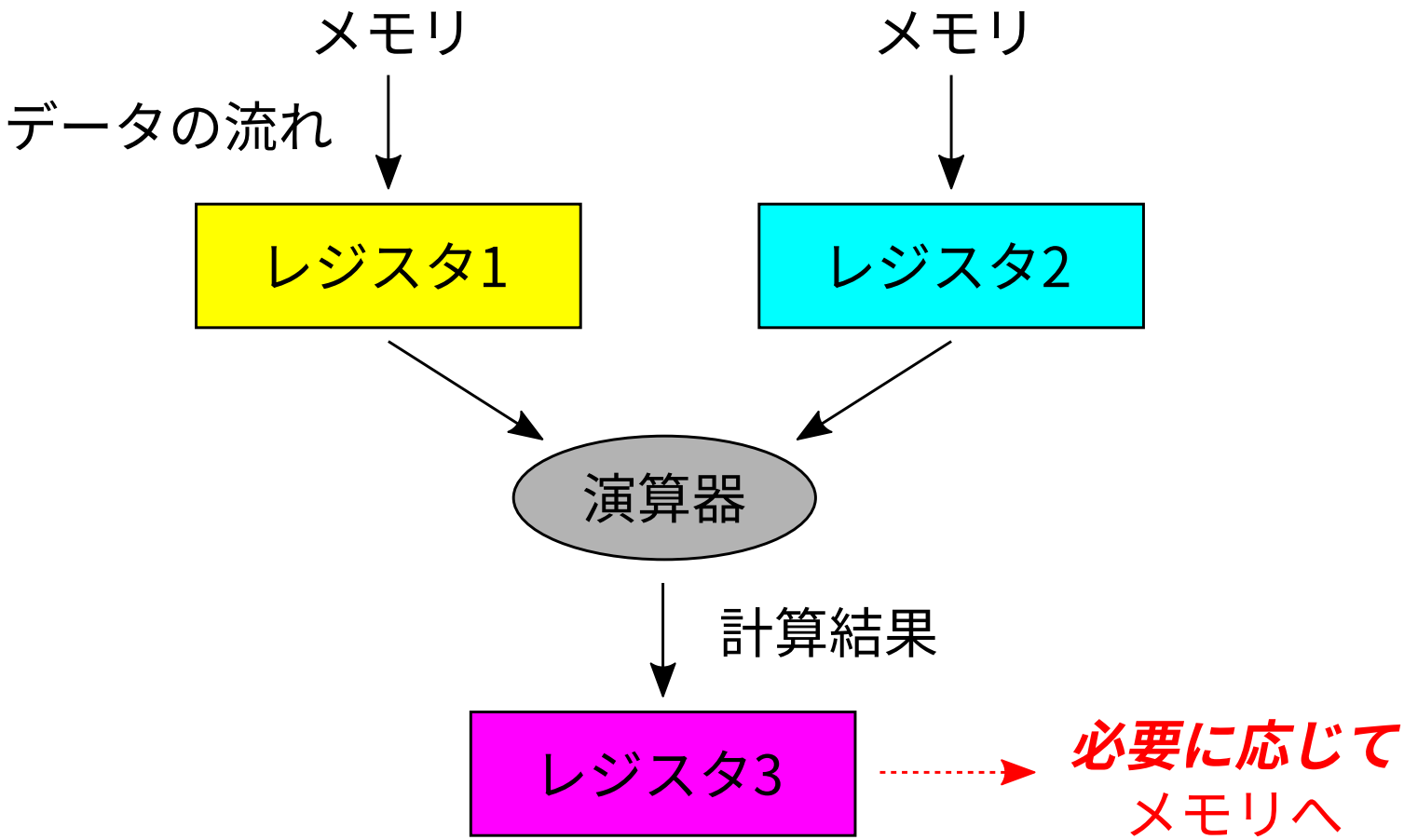
赤字：SSE2 命令

※自動並列化にはなっていない

青字：x87 命令

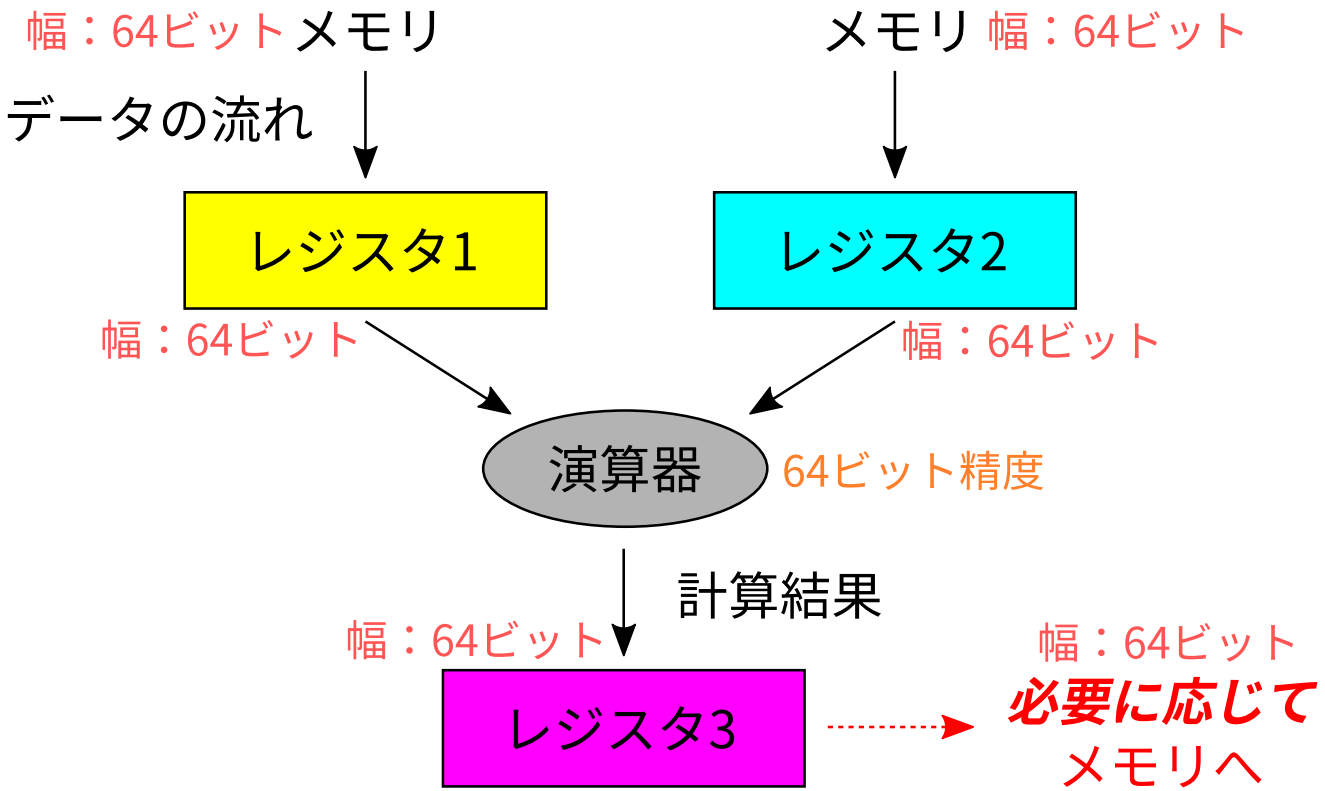
- 同じ double 型の演算でも命令が異なる
- デフォルトのコンパイル・オプションが違うため

アーキテクチャの呪い (3) : CPU 内部



頻繁に更新される変数は
レジスタに居座り続ける

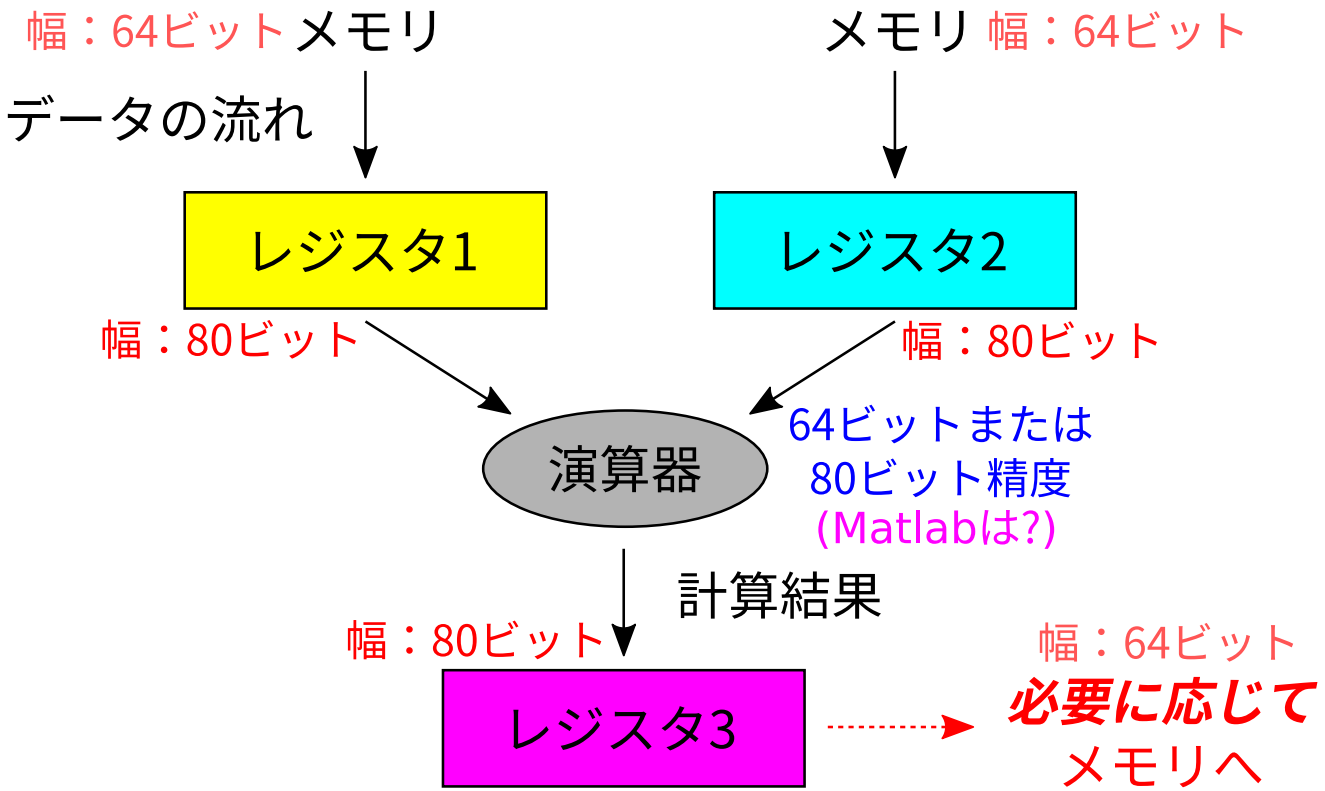
アーキテクチャの呪い (4) : SSE2 の場合



頻繁に更新される変数は
レジスタに居座り続ける

演算精度 = レジスタ幅 = メモリ幅 → コンシステント

アーキテクチャの呪い (5)：x87 の場合



頻繁に更新される変数は
レジスタに居座り続ける

「メモリ ⇔ レジスタ」で結果が変化

アーキテクチャの呪い (6)：初期設計時の懸念のまとめ

- 同じ「正しい」ソースコード
  → コンパイル・オプション次第で「微妙に」異なる結果
- 既存の (古い) 科学ソフトウェアを現代の環境で使用
  - 用途次第では「コンパイル・オプションの追い込み」が必要
  - **Matlab の内部はどうやって確認? (←SNEGRAF に影響)**
- 新しいライブラリ・開発フレームワークの導入
  - Web アプリ化の際に必須
  - 演算精度の追跡は絶望的
  - 新たに生じる問題の例：JavaScript の「整数」：53 ビット

選択肢

1. 潔く全部作り直す
2. “OS + コンパイラ + ソフトウェア” を丸ごと再利用!

SNEGRAF では後者を採用

アーキテクチャの呪い (6)：初期設計時の懸念のまとめ

- 同じ「正しい」ソースコード
  → コンパイル・オプション次第で「微妙に」異なる結果
- 既存の (古い) 科学ソフトウェアを現代の環境で使用
  - 用途次第では「コンパイル・オプションの追い込み」が必要
  - **Matlab の内部はどうやって確認? (←SNEGRAF に影響)**
- 新しいライブラリ・開発フレームワークの導入
  - Web アプリ化の際に必須
  - 演算精度の追跡は絶望的
  - 新たに生じる問題の例：JavaScript の「整数」：53 ビット

選択肢

1. 潔く全部作り直す
2. “OS + コンパイラ + ソフトウェア” を丸ごと再利用!

SNEGRAF では後者を採用

アーキテクチャの呪い (6)：初期設計時の懸念のまとめ

- 同じ「正しい」ソースコード
  → コンパイル・オプション次第で「微妙に」異なる結果
- 既存の (古い) 科学ソフトウェアを現代の環境で使用
  - 用途次第では「コンパイル・オプションの追い込み」が必要
  - **Matlab** の内部はどうやって確認? **(←SNEGRAF** に影響**)**
- 新しいライブラリ・開発フレームワークの導入
  - Web アプリ化の際に必須
  - 演算精度の追跡は絶望的
  - 新たに生じる問題の例：JavaScript の「整数」：53 ビット

選択肢

1. 潔く全部作り直す
2. “OS + コンパイラ + ソフトウェア” を丸ごと再利用!

SNEGRAF では後者を採用

アーキテクチャの呪い (6)：初期設計時の懸念のまとめ

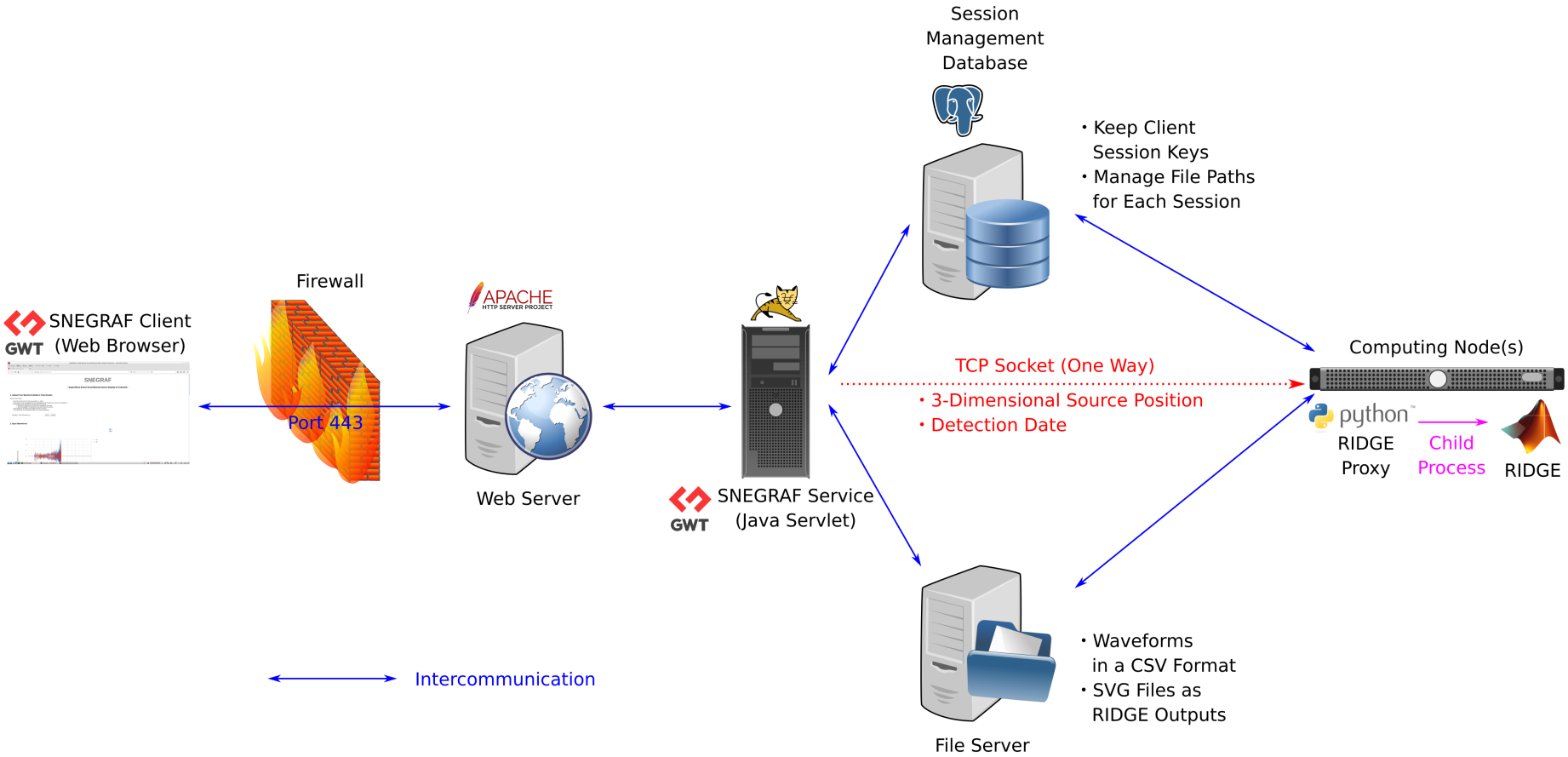
- 同じ「正しい」ソースコード
  → コンパイル・オプション次第で「微妙に」異なる結果
- 既存の (古い) 科学ソフトウェアを現代の環境で使用
  - 用途次第では「コンパイル・オプションの追い込み」が必要
  - **Matlab の内部はどうやって確認? (←SNEGRAF に影響)**
- 新しいライブラリ・開発フレームワークの導入
  - Web アプリ化の際に必須
  - 演算精度の追跡は絶望的
  - 新たに生じる問題の例：JavaScript の「整数」：53 ビット

選択肢

1. 潔く全部作り直す
2. “OS + コンパイラ + ソフトウェア” を丸ごと再利用!

SNEGRAF では後者を採用

SNEGRAF のシステム構成 (全体)



RIDGE パイプラインをハードウェア的に隔離

SNEGRAF のシステム構成 (RIDGE Proxy)

- RIDGE Proxy
  - ソケット通信 (一方向) を利用した簡単なサーバ・プログラム
  - 必要なパラメータを受信後 RIDGE を起動
- セキュリティ・パッチ適用は任意のタイミングで OK!
- 固有部分を修正すれば、他のパイプラインにも対応可能

※インタラクティブでなければ

まとめ

- 既存のソフトウェアの Web アプリ化
  → 計算精度を担保しつつ移植するのは困難
- パイプラインの場合：既存の動作環境を丸ごと隔離
  → 起動パラメータは一方向のソケット通信で受信
  →OS やライブラリの更新に追随する必要はない
- SNEGRAF の開発により必要最低限の仕組みは完成