多倍長数値計算の基礎と応用

静岡理工科大学
幸谷智紀
http://na-inet.jp/

第 4 回数理科学研究会@岐阜県高山市
2015 年 1 月 19 日 (月)

概要

- 自己紹介
- 多倍長数値計算＝多倍長浮動小数点演算の基本
- 多倍長数値計算の高速化
- 連立一次方程式の解計算の高速化
- まとめと今後の課題

- 自己紹介
- 多倍長数値計算＝多倍長浮動小数点演算の基本
- 多倍長数値計算の高速化
- 連立一次方程式の解計算の高速化
- まとめと今後の課題

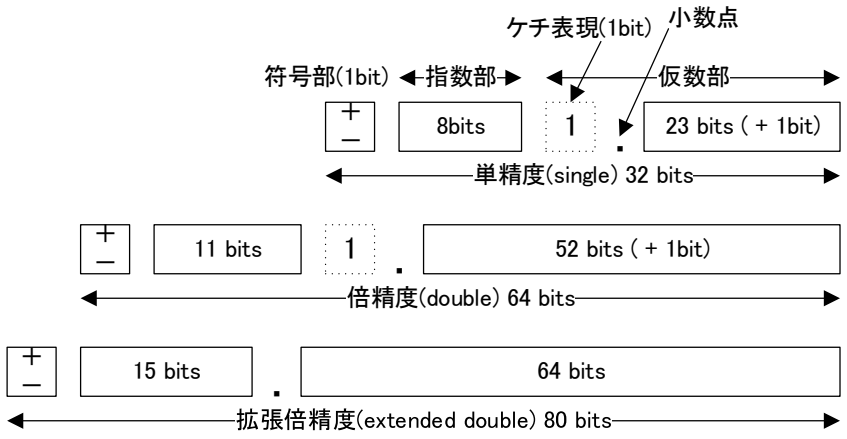
自己紹介と研究テーマ

- 数値計算，特に丸め誤差解析に興味
  - 悪条件問題に対して単純なアプローチを取りたい
  - 数字を眺めるのが好き＆コンピュータを使い倒すのが好き

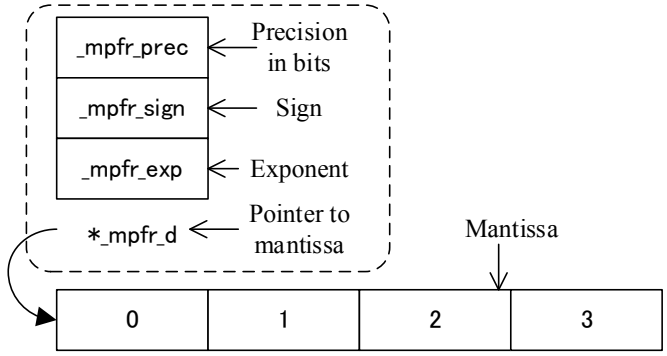
  →多倍長浮動小数点演算を用いた数値計算の研究
- 多倍長浮動小数点演算には MPFR/GMP を利用→多倍長数値計算ライブラリ BNCpack を制作
  (cf. 応用数理, Vol, 21, pp.197-206, 2011)
- 近年の研究テーマ１「倍精度・多倍長精度混合精度反復改良法の陰的 Runge-Kutta 法への応用」
  (cf. arXiv preprint arXiv:1306.2392, 日本応用数理学会論文誌 Vol.19, No.3, pp.313-328, 2009)
- 近年の研究テーマ２「Strassen のアルゴリズムを用いた多倍長行列積の高速化と直接法への応用」
  (cf. JSIAM Letters, Vol.6, pp.81-84, 2014)

- 自己紹介
- 多倍長数値計算＝多倍長浮動小数点演算の基本
- 多倍長数値計算の高速化
- 連立一次方程式の解計算の高速化
- まとめと今後の課題

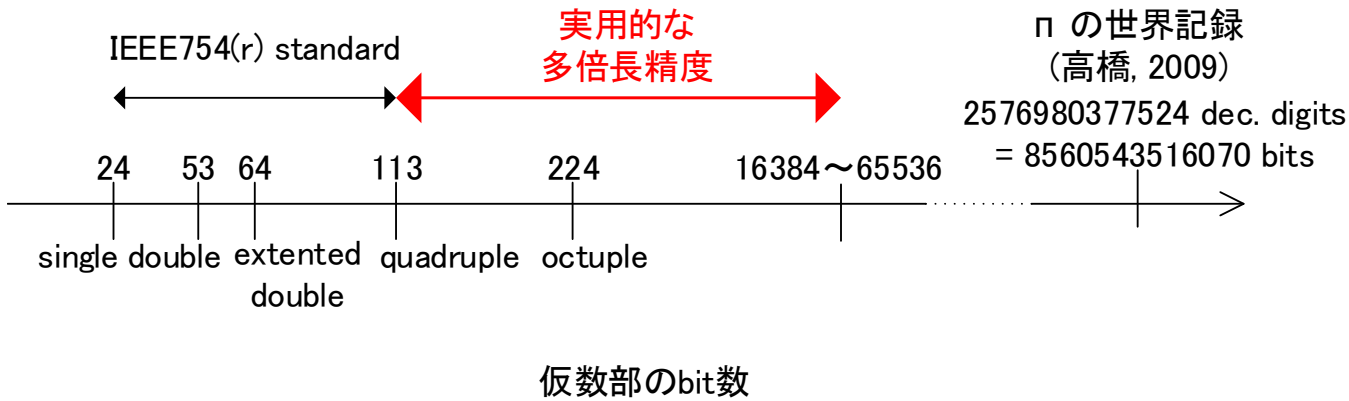
浮動小数点数の形式と実用的な多倍長精度の範囲



IEEE754-1985 形式：ハードウェア (CPU, GPU) が直接演算処理



多倍長精度浮動小数点数 (mpfr_t 形式)：ソフトウェアで演算処理



「実用的な多倍長精度」の意味＝「ユーザが要求する計算精度」と「四則演算が 1 秒未満で完了する計算精度」

多倍長浮動小数点演算の実装方法による分類

「多倍長浮動小数点数」の定義：IEEE754 単精度・倍精度より仮数部 (小数部) の桁数が多い浮動小数点数

固定長 vs. 可変長

一度確保された浮動小数点数の仮数部の長さを変えられるかどうか？

固定長 DD(4 倍精度)・QD(8 倍精度)，exflib

可変長 MPFR/GMP, mpf(GMP), ARPREC(?)

IEEE745 倍精度演算ベース vs. 整数演算ベース

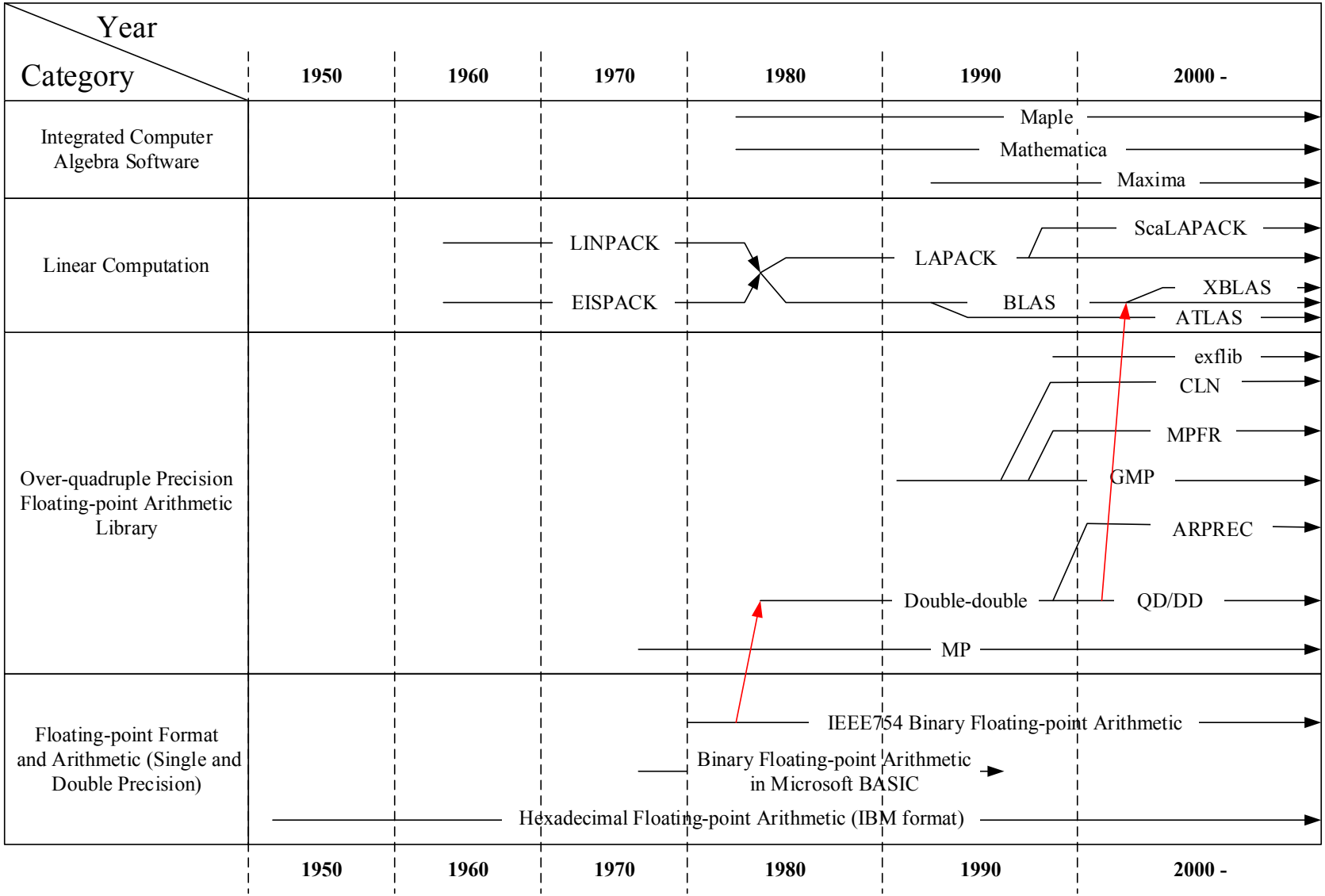
多倍長演算をどの演算の組み合わせで実現するか？

倍精度演算ベース DD・QD, ARPREC (by Baily)

整数演算ベース exflib, MPFR/GMP, mpf(GMP)，その他 (GMP ベースのものがかなりある)

cf. MPACK (by 中田真秀 (理研)) = LAPACK / (DD・QD + MPFR) ⋯ 多倍長線型計算ライブラリ

数学ソフトウェアの歴史

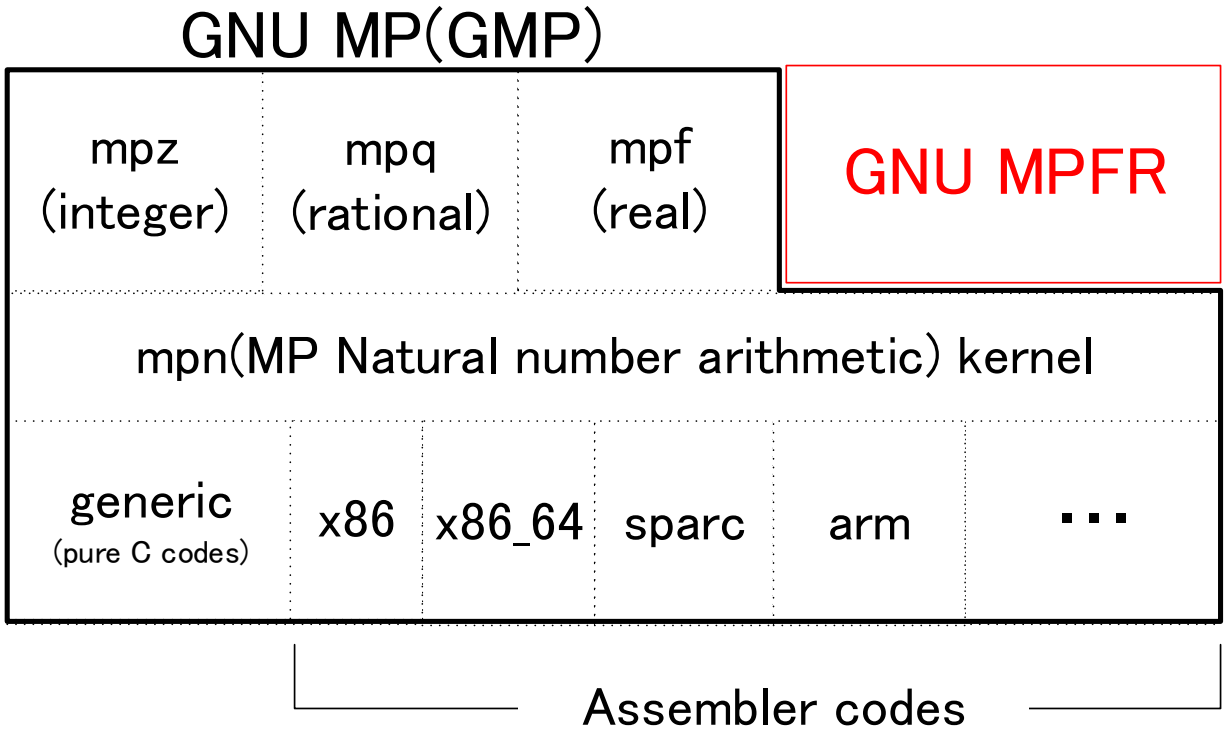


基本的な多倍長精度演算ライブラリはここ 20 年ぐらいで固定化。

MPFR/GMP とは？

GMP (GNU MP) 多倍長自然数演算 (mpn) カーネルを土台にした任意精度ライブラリ。整数 (mpz)，有理数 (mpq)，浮動小数点数 (mpf) をサポート。Version 6.0.0a が最新。`http://gmplib.org/`

MPFR (GNU MPFR) GMP の mpn カーネルを土台とした，IEEE754 互換の浮動小数点演算ライブラリ。Version 3.1.2 が最新。`http://www.mpfr.org/`



多倍長演算ライブラリの速度比較 (単位：ミリ秒)

http://www.mpfr.org/mpfr-current/timings.html

100 digits	Maple	Mathematica	Sage	MPF	MPFR	Pari	NTL	CLN	ARB	MPFI
mult	0.0020	0.0006	0.00047	**0.00011**	0.00014	0.00012	0.000367	0.00018	0.000139	0.000265
sqr			0.00045	**0.00009**	0.00011	0.00011		0.00015	0.000120	0.000210
div	0.0029	0.0017	0.00067	0.00031	**0.00030**	0.00034	0.00070	0.00049	0.00047	0.00068
sqrt	0.032	0.0018	0.00106	0.00056	0.00050	**0.00049**	0.00442	0.00067	0.00060	0.00098
exp	0.070	0.019	0.0101	na	0.0084	0.0106	0.069	0.0197	**0.0019**	0.0143
log	0.100	0.028	0.0171	na	0.0121	0.0117	0.386	0.0278	**0.0020**	0.0208
sin	0.131	0.017	0.0098	na	0.0083	0.0095	0.074	0.0253	**0.0024**	0.0177
cos	0.119	0.018	0.0068	na	0.0059	0.0085	0.082	0.0212	**0.0021**	0.0139
acos	0.450	0.053	0.052	na	0.049	0.028	na	0.033	**0.0045**	0.088
atan	0.280	0.048	0.046	na	0.043	0.026	na	0.028	**0.0018**	0.075

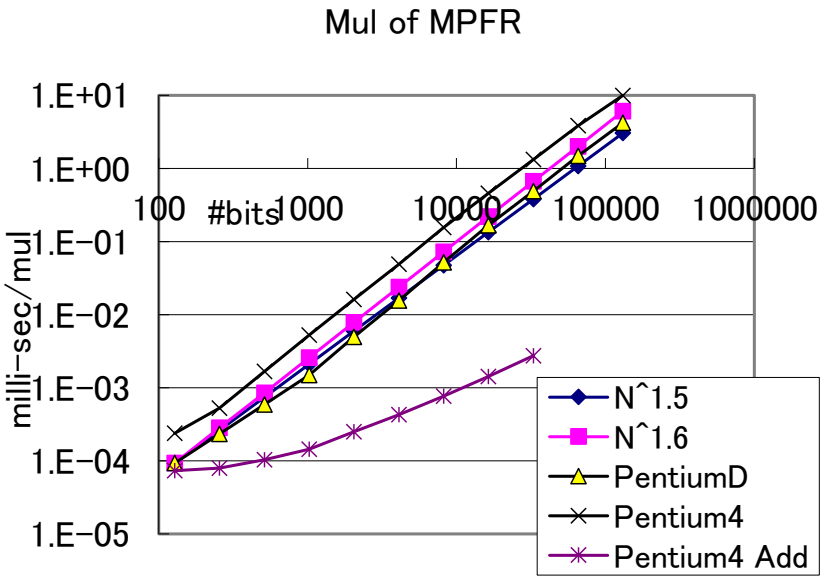
10000 digits	Maple	Mathematica	Sage	MPF	MPFR	Pari	NTL	CLN	ARB	MPFI
mult	0.80	0.28	0.092	0.107	**0.091**	0.108	0.508	0.106	0.094	0.188
sqr			0.063	0.076	**0.060**	0.076		0.076	0.063	0.123
div	0.80	0.56	0.196	0.198	**0.181**	0.264	1.662	0.503	0.256	0.364
sqrt	3.70	0.36	0.193	0.178	**0.175**	0.176	20.48	0.295	0.177	0.347
exp	50.0	17.6	8.2	na	8.5	12.6	1560	13.6	**5.4**	17.3
log	20.0	15.9	7.3	na	**6.9**	8.2	16080	16.7	7.0	13.9
sin	93.0	44.4	**15.3**	na	15.4	21.7	1650	17.6	15.4	31.7
cos	92.0	44.4	**15.3**	na	15.4	21.6	7710	16.5	16.5	31.9
acos	87.0	91.2	15.1	na	**15.0**	31.1	na	28.6	17.2	30.0
atan	82.0	87.2	**13.7**	na	**13.7**	31.0	na	27.0	14.2	27.3

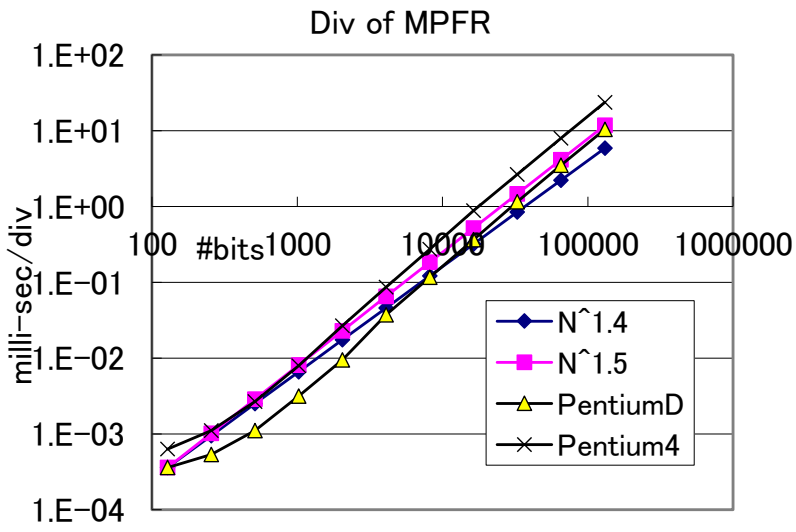
→ MPFR/GMP は最高速の項目が多い。

MPFR/GMP の速さの秘訣

高速な GMP の mpn カーネルのおかげ。特に，乗算，除算，平方の計算の高速化がカギ。

- 桁数小：basecase アルゴリズム→ Karatsuba アルゴリズム→ Toom-Cook アルゴリズム→ FFT：桁数大
- CPU アーキテクチャごとにメモリ操作，ビット演算，SIMD 命令をアセンブラ化





→大体 $O(N^{1.4})$〜$O(N^{1.6})$ 程度で収まっている。

BNCpack について

- BNCpack は，Basic Numerical Computation PACKage の略称
- 2001 年から開発を開始。GMP の mpf_t 型もしくは MPFR/GMP の mpfr_t 型にも対応
- 使用言語は全て C (C++にあらず)。MPI にも対応 (後述)

現在提供されているのは次の機能

1. 複素数演算
2. 基本線型計算 (ベクトル，正方密行列の和・差・内積 (積)・スカラー倍など)
3. 連立一次方程式 (直接法，反復法，クリロフ部分空間法)
4. 行列の固有値・固有ベクトル計算 (べき乗法・逆べき乗法，QR 法)
5. 非線型方程式 (Newton 法，準 Newton 法, Regula-Falsi 法)
6. 代数方程式 (DKA 法)
7. 数値積分 (台形則，Romberg 積分)
8. 常微分方程式の初期値問題 (陽的・陰的ルンゲ　クッタ法，補外法)
9. その他 (スタックなど)

数値実験

$A = [a_{ij}]$, $B = [b_{ij}]$ は下記のものを使用し，$C := AB$ を計算。

$$a_{ij} = \sqrt{5}\,(i + j - 1),\ b_{ij} = \sqrt{3}\,(n - i + 1).$$

単純行列積計算

$$c_{ij} := \sum_{k=1}^{l} a_{ik} b_{kj}$$

ブロック行列積計算

キャッシュヒット率を上げるための工夫。

$$A = [A_{ik}], B = [B_{kj}]\,(1 \le i \le M, 1 \le k \le L, 1 \le j \le N)$$

A, B をブロック行列化して，行列 $C = [C_{ij}]$ を次のように計算。

$$C_{ij} := \sum_{k=1}^{L} A_{ik} B_{kj}$$

計算時間 (秒): 128bits 計算 vs. Intel Math Kernel(倍精度)

H/W Intel Core i7 3850 (3.6 GHz), 64 GB RAM

S/W Scientific Linux 6.3 x86_64, Intel C Compiler Ver. 13.0.1, BNCpack ver. 0.8, MPFR 3.1.2, GMP 5.1.3

$m \times n$	Simple	Block(16)	Block(32)	Block(64)	Double
255 × 255	1.06	1.20	1.22	1.24	
256 × 256	1.25	1.22	1.22	1.25	
257 × 257	1.04	1.25	1.28	1.37	
511 × 511	9.60	9.71	9.61	10.02	
512 × 512	10.83	9.70	9.68	9.96	
513 × 513	10.02	9.89	9.97	10.44	
1023 × 1023	107.78	77.63	77.80	79.36	0.084
1024 × 1024	213.09	77.77	77.72	79.51	0.083
1025 × 1025	94.62	78.92	78.41	81.48	0.087
2047 × 2047	756.81	627.75	619.21	648.31	0.658
2048 × 2048	1679.04	624.86	618.87	639.71	0.653
2049 × 2049	632.74	623.24	625.69	640.84	0.675

計算時間: 1024bits 計算

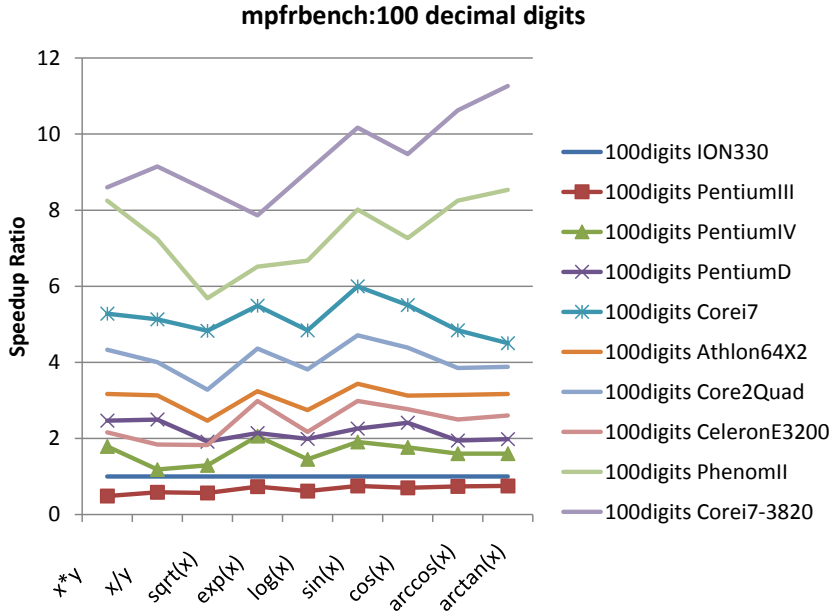
$m \times n$	Simple	Block(16)	Block(32)	Block(64)
255 × 255	5.5	5.46	5.47	5.54
256 × 256	5.77	5.53	5.53	5.64
257 × 257	5.68	5.61	5.72	5.80
511 × 511	45.73	43.81	43.96	44.51
512 × 512	49.71	44.37	44.31	44.20
513 × 513	46.58	44.37	44.77	45.12
1023 × 1023	372.10	352.79	354.37	353.15
1024 × 1024	463.79	356.10	356.85	355.99
1025 × 1025	385.24	356.58	357.17	361.24
2047 × 2047	3122.48	2829.43	2820.16	2833.66
2048 × 2048	3754.89	2845.70	2824.34	2859.24
2049 × 2049	2933.26	2829.95	2835.54	2859.66

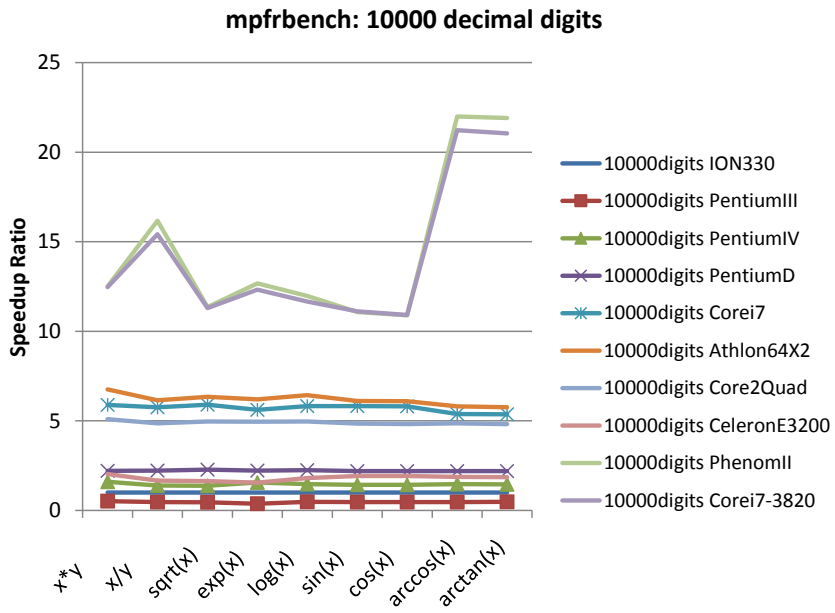
→ 2048bits 以上になると，単純行列積計算の方が高速になる
(alloc, free の影響大？)

- 自己紹介
- 多倍長数値計算＝多倍長浮動小数点演算の基本
- 多倍長数値計算の高速化
- 連立一次方程式の解計算の高速化
- まとめと今後の課題

高速化＝並列化手法

CPU の性能は，動作周波数以上に上がり続けている。





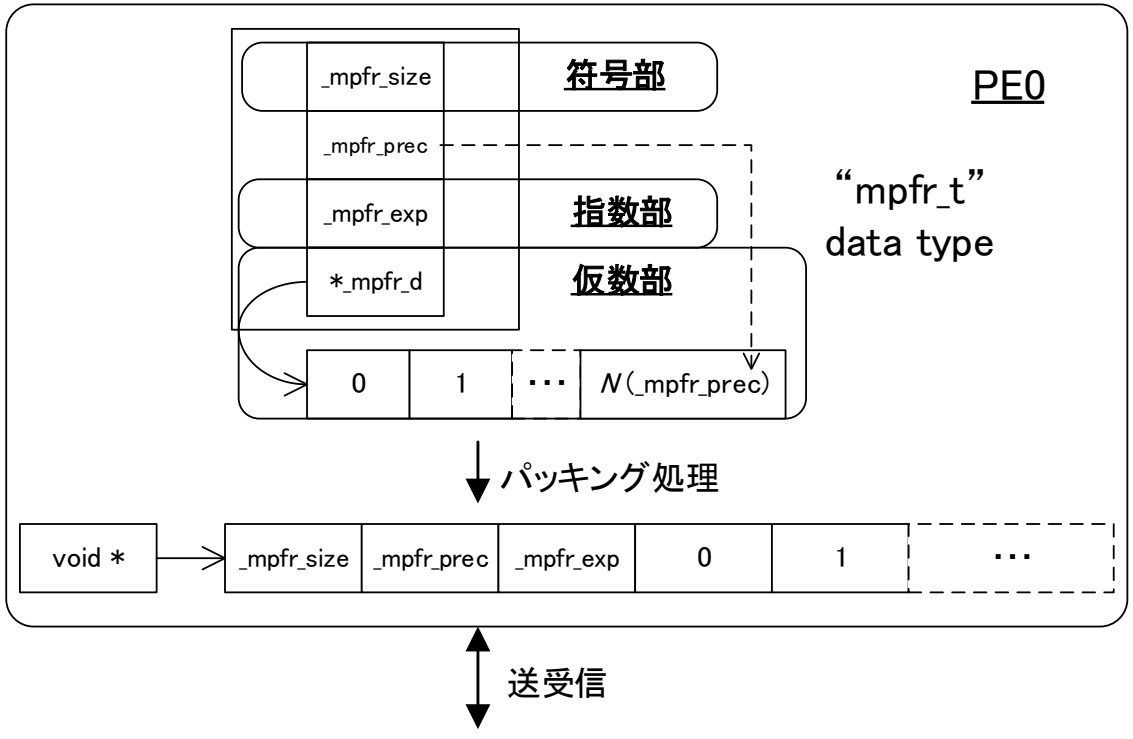
1CPU(1 core) における高速化はほぼ限界に達している
→高速化するには並列処理するしかない。

分散メモリ　MPI → MPIBNCpack

共有メモリ　OpenMP(マルチコア CPU) + BNCpack,
CUDA(NVIDIA GPU) → CUMP

分散メモリ環境における並列化：MPIBNCpack

分散メモリに MPFR/GMP or GMP を対応させるためのライブラリ (MPIGMP) を実装し，密行列演算を高速化。



詳細は Tutorial を参照　→　`http://na-inet.jp/tutorial/`

共有メモリ環境における並列化：OpenMP(マルチコアCPU)，GPU の利用

CUMP (`http://www.hpcs.cs.tsukuba.ac.jp/~nakayama/cump/`) の特徴：

- GMP の mpf_t ベースの実装
- mpn カーネルは generic C を利用
- 自然数乗算は basecase アルゴリズムのみサポート（最大のボトルネック）

計算環境：

ハードウェア (H/W) Intel Xeon E5-2620 v2(2.10GHz) × 2 = 12 cores, NVIDIA Tesla K20

ソフトウェア (S/W) CentOS 6.5 x86_64, gcc-4.4.7, Intel C compiler 13.1.3, GMP 6.0.0a, MPFR 3.1.2, CUDA 6.5

単純行列積計算を，CPU:12 threads, GPU: 8× 8 = 64 threads で実行

12 cores CPU vs. Tesla K20 GPU

計算時間の比率：CPU の計算時間 / GPU の計算時間

CPU/GPU	128 × 128	256 × 256	512 × 512	1024 × 1024
128bits	0.87	1.15	2.09	8.44
256	0.56	0.88	2.27	7.67
512	0.41	0.28	1.60	3.84
1024	0.12	0.35	1.20	1.93
2048	0.09	0.39	0.82	0.96
4096	0.17	0.46	0.63	0.66
8192	0.25	0.42	0.46	
16384	0.25	0.31	0.32	
32768	0.18	0.21		

→ CPU のコア数が多いと GPU の方が不利な領域が増える。

→ GPU のグローバルメモリの量で計算できる精度・行列サイズが決まってしまう。

- 自己紹介
- 多倍長数値計算＝多倍長浮動小数点演算の基本
- 多倍長数値計算の高速化
- 連立一次方程式の解計算の高速化
- まとめと今後の課題

連立一次方程式の解計算の高速化

解くべき連立一次方程式：

$$\text{[真の方程式]} \quad A\mathbf{x} = \mathbf{b}$$

$$\text{ここで } A \in \mathbb{R}^{n\times n}, \mathbf{b}, \mathbf{x}^i n\mathbb{R}^n$$

↓誤差 $E(\widetilde{A})$, $E(\widetilde{\mathbf{b}})$ が混入→ $E(\widetilde{\mathbf{x}})$ に影響↓

$$\text{[誤差混入後]} \quad \widetilde{A}\widetilde{\mathbf{x}} = \widetilde{\mathbf{b}}$$

$$\text{ここで}\widetilde{A} = A + E(\widetilde{A}), \widetilde{\mathbf{b}} = \mathbf{b} + E(\widetilde{\mathbf{b}}), \widetilde{\mathbf{x}} = \mathbf{x} + E(\widetilde{\mathbf{x}})$$

条件数 $\kappa(A) = \|A\|\|A^{-1}\|$ の大きさで数値解に含まれる誤差が変化する。

$$\frac{\|E(\widetilde{\mathbf{x}})\|}{\|\mathbf{x}\|} \leq \frac{\kappa(A)}{1 - \dfrac{\|E(\widetilde{A})\|}{\|A\|}} \cdot \left(\frac{\|E(\widetilde{A})\|}{\|A\|} + \frac{\|E(\widetilde{\mathbf{b}})\|}{\|\mathbf{b}\|} \right)$$

良条件問題　混合精度反復改良法

悪条件問題　Strassen アルゴリズムによる LU 分解の高速化

混合精度反復改良法

反復改良法は 1967 年に Moler が提案.
n 次元方程式

$$\mathbf{f}(\mathbf{x}) = 0,\ \mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n \tag{1}$$

を解くための Newton 法がベース．(1) が線型方程式

$$\mathbf{f}(\mathbf{x}) = A\mathbf{x} - \mathbf{b}$$

であれば，Jacobi 行列は定係数行列 A となるので，次のようなアルゴリズムで反復を進めることになる $(L >> S)$.

$$[L \text{ 桁計算}] \quad \mathbf{r}_k := \mathbf{b} - A\mathbf{x}_k \tag{2}$$

$$[S \text{ 桁計算}] \quad \text{Solve } A\mathbf{z}_k = \mathbf{r}_k \text{ for } \mathbf{z}_k \tag{3}$$

$$[L \text{ 桁計算}] \quad \mathbf{x}_{k+1} := \mathbf{x}_k + \mathbf{z}_k \tag{4}$$

理論的には反復する必要はないが，現実に有限桁の浮動小数点数点演算を使用すると丸め誤差の影響で残差 $\mathbf{r}_k$ がゼロにならないため，これを最小化するように複数回の反復が行われる．そのため，近似解 $\mathbf{x}_k$ の精度を上げるためには，残差の計算は高精度で行う必要がある.

直接法を使用した混合精度反復改良法

以上をアルゴリズムの形でまとめると，(2)〜(4) は以下のようになる．ここで，$A^{[S]}$, $\mathbf{b}^{[L]}$ はそれぞれ S 桁，L 桁の浮動小数点数で表現した行列・ベクトルを意味する．

$A^{[L]} := A$, $A^{[S]} := A^{[L]}$, $\mathbf{b}^{[L]} := \mathbf{b}$, $\mathbf{b}^{[S]} := \mathbf{b}^{[L]}$

$A^{[S]} := P^{[S]} L^{[S]} U^{[S]}$

Solve $(P^{[S]} L^{[S]} U^{[S]})\mathbf{x}_0^{[S]} = \mathbf{b}^{[S]}$ for $\mathbf{x}_0^{[S]}$

$\mathbf{x}_0^{[L]} := \mathbf{x}_0^{[S]}$

For $k = 0, 1, 2, ...$

- $\mathbf{r}_k^{[L]} := \mathbf{b}^{[L]} - A\mathbf{x}_k^{[L]}$
- $\mathbf{r}_k^{[S]} := \mathbf{r}_k^{[L]}$
- Solve $(P^{[S]} L^{[S]} U^{[S]})\mathbf{z}_k^{[S]} = \mathbf{r}_k^{[S]}$ for $\mathbf{z}_k^{[S]}$
- $\mathbf{z}_k^{[L]} := \mathbf{z}_k^{[S]}$
- $\mathbf{x}_{k+1}^{[L]} := \mathbf{x}_k^{[L]} + \mathbf{z}_k^{[L]}$
- Exit if $\|\mathbf{r}_k^{[L]}\|_2 \leq \sqrt{n}\ \varepsilon_R\ \|A\|_F \|\mathbf{x}_k^{[L]}\|_2 + \varepsilon_A$

収束条件

$$\frac{\rho_F(n)\kappa(A)\varepsilon_S}{1-\psi_F(n)\kappa(A)\varepsilon_S} < 1 \text{ かつ } \alpha_F < 1 \tag{5}$$

であれば

$$\lim_{k\to\infty} \|\mathbf{x}-\mathbf{x}_k\| \leq \frac{\beta_F}{1-\alpha_F}\|\mathbf{x}\| \tag{6}$$

となり，ノルム相対誤差が $\beta_F/(1-\alpha_F)$ 程度まで小さくなることが期待できる.

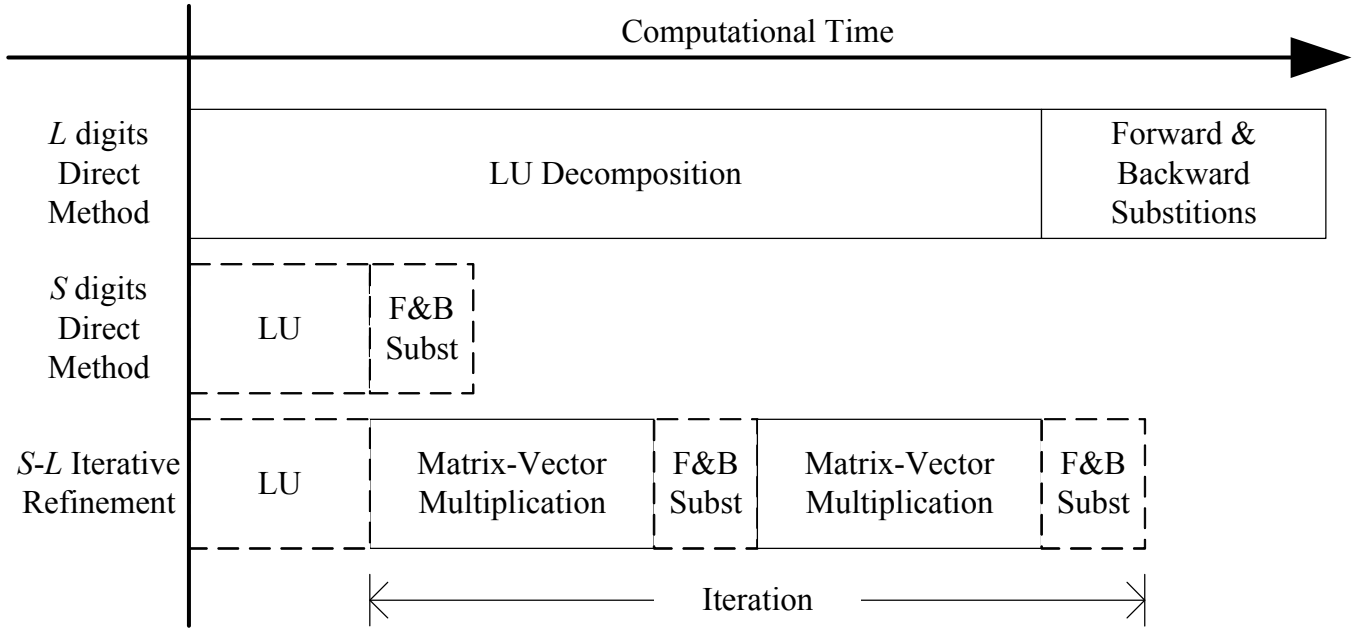
(cf. A.Buttari, J.Dogarra, Julie Langou, Julien Langou, P.Luszczek, and J.Karzak, "Mixed precision iterative refinement techniques for the solution of dense linear system", The International Journal of High Performance Computing Applications, Vol. 21, No. 4, pp. 457–466, 2007.)

→ S-L 桁混合精度反復改良法が収束するためには,

$$\kappa(A)\varepsilon_S << 1 \tag{7}$$

でなければならない。ε_S, ε_L は S 桁,L 桁計算のマシンイプシロン。

混合精度反復改良法の計算時間



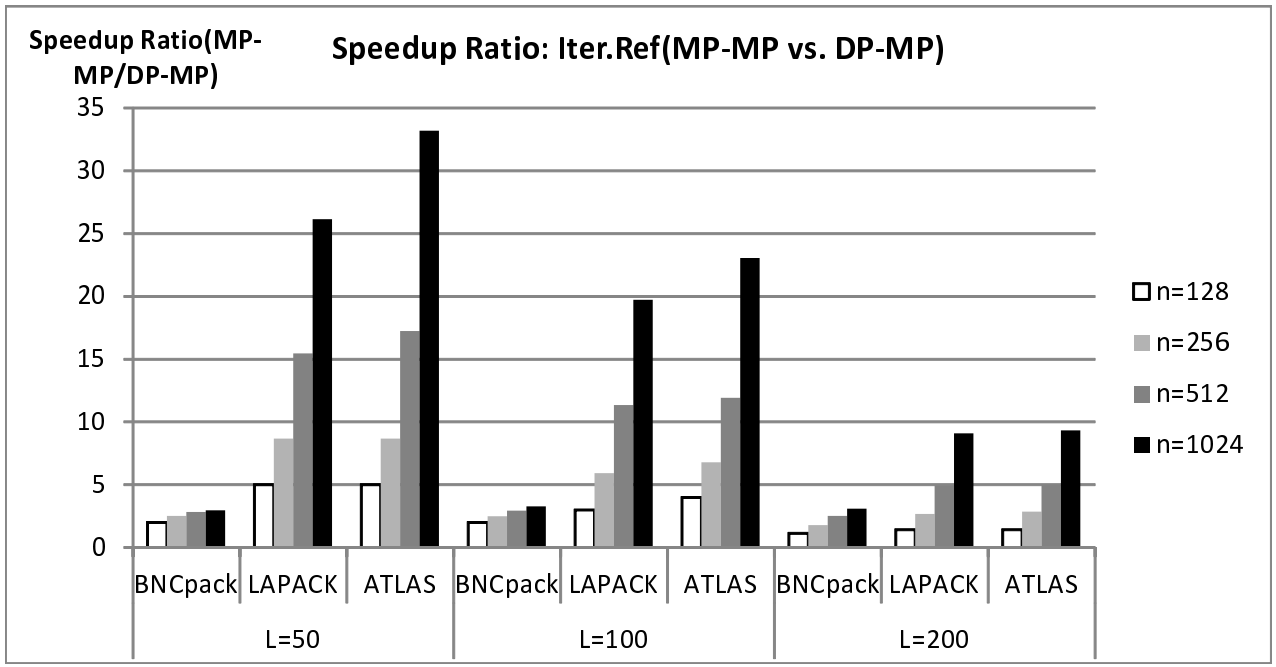
条件数に応じて下記の 3 種の組み合わせを用いる。

1. $\kappa(A) < 10^7 \implies$ 単精度 $(S = 7)$-倍精度 $(L = 15)$: SP-DP 型 (オリジナル)
2. $\kappa(A) < 10^{15} \implies$ 倍精度 $(S = 15)$-4 倍精度 or 多倍長精度 $(L > 30)$: DP-MP 型
3. $\kappa(A) > 10^{15} \implies$ 4 倍精度 or 多倍長精度 - 8 倍精度 or 多倍長精度 : MP-MP 型

DP-MP 型混合精度反復改良法のベンチマークテスト

H/W AMD Athlon64X2 3800+, 4GB

S/W CentOS 5.2 x86_64, GCC 4.1.2, MPFR 2.3.2/GMP 4.2.1 + BNCpack 0.7b, LAPACK 3.2, ATLAS 3.8.3



応用 (1/6): n 次元常微分方程式の初期値問題

解くべき n 次元常微分方程式 (ODE) の初期値問題

$$\begin{cases} \frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}) \ \in \mathbb{R}^n \\ \mathbf{y}(t_0) = \mathbf{y}_0 \end{cases} \tag{8}$$

積分区間:$[t_0, \alpha]$

ここで m 段 IRK 公式を規定する係数 $c_1, ..., c_m$, $a_{11}, ..., a_{mm}$, $b_1, ..., b_m$ は m 段 $2m$ 次の Gauss 型公式を利用。

$$\begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1m} \\ \vdots & \vdots & & \vdots \\ c_m & a_{m1} & \cdots & a_{mm} \\ \hline & b_1 & \cdots & b_m \end{array} = \begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b}^T \end{array} \tag{9}$$

応用 (2/6): m 段陰的 Runge-Kutta 法のアルゴリズム概要

積分区間の離散化：t_0, $t_1 := t_0 + h_0$, ..., $t_{k+1} := t_k + h_k$...
t_k から t_{k+1} への近似解 $\mathbf{y}_{k+1} \approx \mathbf{y}(t_{k+1})$ を求めるため，次の 2 段階の計算を行う。

(A) 次の非線型方程式を $\mathbf{Y} = [Y_1 \ ... \ Y_m]^T \in \mathbb{R}^{mn}$ について解く。

$$\left\{ \begin{array}{rcl} Y_1 & = & \mathbf{y}_k + h_k \sum_{j=1}^{m} a_{1j} \mathbf{f}(t_k + c_j h_k, Y_j) \\ & \vdots & \\ Y_m & = & \mathbf{y}_k + h_k \sum_{j=1}^{m} a_{mj} \mathbf{f}(t_k + c_j h_k, Y_j) \end{array} \right.$$

$$\Updownarrow$$

$$\mathbf{F}(\mathbf{Y}) = 0 \tag{10}$$

(B) 求めた $\mathbf{Y}$ を用いて次の近似解 $\mathbf{y}_{k+1}$ を求める。

$$\mathbf{y}_{k+1} := \mathbf{y}_k + h_k \sum_{j=1}^{m} b_j \mathbf{f}(t_k + c_j h_k, Y_j)$$

応用 (3/6):SPARK3 型リダクション (1/2)

W 変換 (Hairer & Wanner) を用いた実三重対角化 (Gauss 型の場合)

$$X = W^T BAW = \begin{bmatrix} 1/2 & -\zeta_1 & & & \\ \zeta_1 & 0 & \ddots & & \\ & \ddots & \ddots & -\zeta_{m-2} & \\ & & \zeta_{m-2} & 0 & -\zeta_{m-1} \\ & & & \zeta_{m-1} & 0 \end{bmatrix}$$

ここで

$$W = [w_{ij}] = [\tilde{P}_{j-1}(c_i)] \ (i, j = 1, 2, ..., m)$$

$(\tilde{P}_s(x)$: s-th shifted Legendre polynomial)

$$\zeta_i = \left(2\sqrt{4i^2 - 1}\right)^{-1} \quad (i = 1, 2, ..., m-1)$$

$$B = \mathrm{diag}(\mathbf{b}), \ I_m = W^T BW = \mathrm{diag}(1\ 1\ \cdots\ 1)$$

応用 (4/6):SPARK3 型リダクション (2/4)

よって，簡易 Newton 法の連立一次方程式の係数行列は

$$(W^T B \otimes I_n)(I_m \otimes I_n - h_k A \otimes J)(W \otimes I_n)$$

$$= I_m \otimes I_n - h_k X \otimes J = \begin{bmatrix} E_1 & F_1 & & & \\ G_1 & E_2 & F_2 & & \\ & \ddots & \ddots & \ddots & \\ & & G_{m-2} & E_{m-1} & F_{m-1} \\ & & & G_{m-1} & E_m \end{bmatrix}$$

となる。ここで

$$E_1 = I_n - \frac{1}{2} h_k J,\ E_2 = \cdots = E_s = I_n$$

$$F_i = h_k \zeta_i J,\ G_i = -h_k \zeta_i J\ (i = 1, 2, ..., m-1)$$

となる。

応用 (5/6):SPARK3 型リダクション + 簡易 Newton 法 + 混合精度反復改良法のアルゴリズム

1. $\mathbf{Y}_{-1} := [\mathbf{y}_k\ \mathbf{y}_k\ \ldots\ \mathbf{y}_k]^T \in \mathbb{R}^{mn}$
2. For $l = 0, 1, 2, \ldots$ ⋯ 簡易 Newton 法

   $\mathbf{Y}_l := [Y_1^{(l)}\ Y_2^{(l)}\ \ldots\ Y_m^{(l)}]^T$ ここで $Y_i^{(l)} = \mathbf{y}_0 + h_k \sum_{j=1}^m a_{ij}\mathbf{f}(t_k + c_i h_k, Y_i^{(l-1)})$

   $C := I_m \otimes I_n - h_k X \otimes J$, $\mathbf{d} := (W^T B \otimes I_n)(-\mathbf{F}(\mathbf{Y}_l))$

   (S) Solve $C\mathbf{x}_0 = \mathbf{d}$ for $\mathbf{x}_0$

   For $\nu = 0, 1, 2, \ldots$ ⋯ 混合精度反復改良法

   $\mathbf{r}_\nu := \mathbf{d} - C\mathbf{x}_\nu$

   (S) $\mathbf{r}'_\nu := \mathbf{r}_\nu / \|\mathbf{r}_\nu\|$

   (S) Solve $C\mathbf{z} = \mathbf{r}'_\nu$ for $\mathbf{z}$

   $\mathbf{x}_{\nu+1} := \mathbf{x}_\nu + \|\mathbf{r}_\nu\|\mathbf{z}$

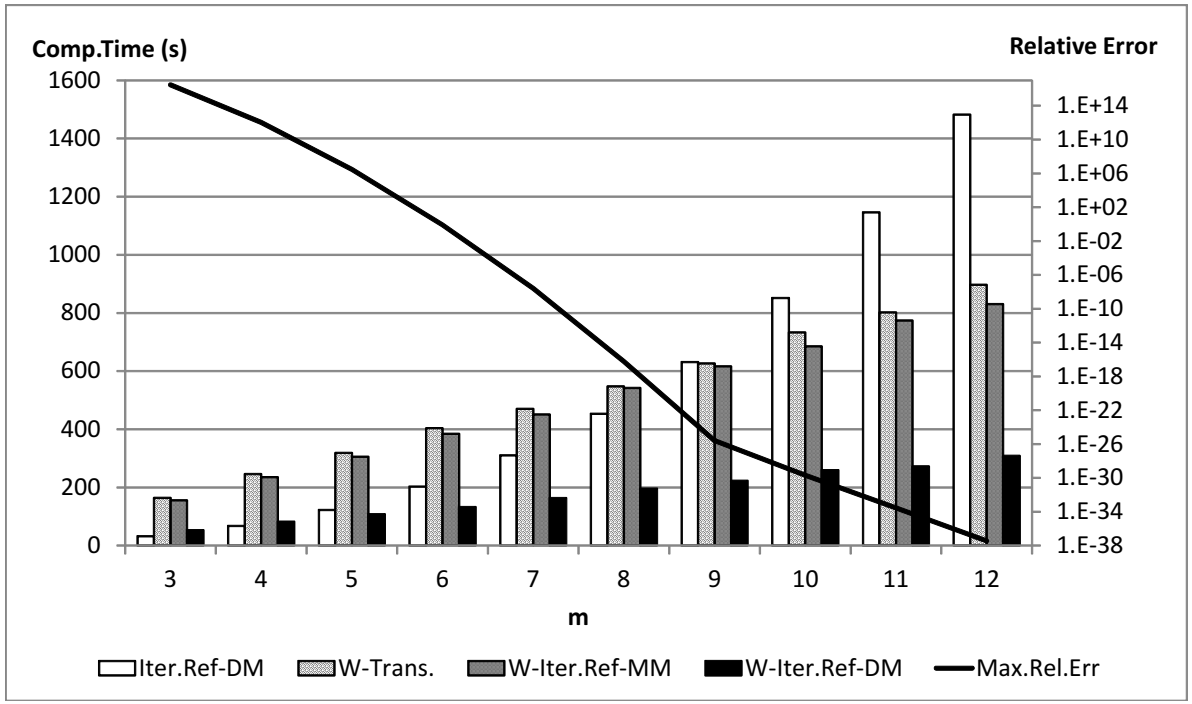
   Check convergence $\Rightarrow \mathbf{x}_{\nu_{stop}}$

   $\mathbf{Y}_{l+1} := \mathbf{Y}_l + (W \otimes I_n)\mathbf{x}_{\nu_{stop}}$

   Check convergence $\Rightarrow \mathbf{Y}_{l_{stop}}$
3. $\mathbf{Y} := \mathbf{Y}_{l_{stop}} = [Y_1\ Y_2\ \ldots\ Y_m]^T$
4. $\mathbf{y}_{k+1} := \mathbf{y}_k + h_k \sum_{j=1}^m b_j \mathbf{f}(t_k + c_j h_k, Y_j)$

応用 (6/6)：フル陰的 Runge-Kutta 法 (Gauss 型) の高速化

- 128 次元定係数線型 ODE の性能評価 (10 進約 50 桁)
- Intel Core i7 920, 8GB RAM, CentOS 5.6 x86_64, gcc 4.1.2, MPFR 3.1.1/GMP 5.0.5

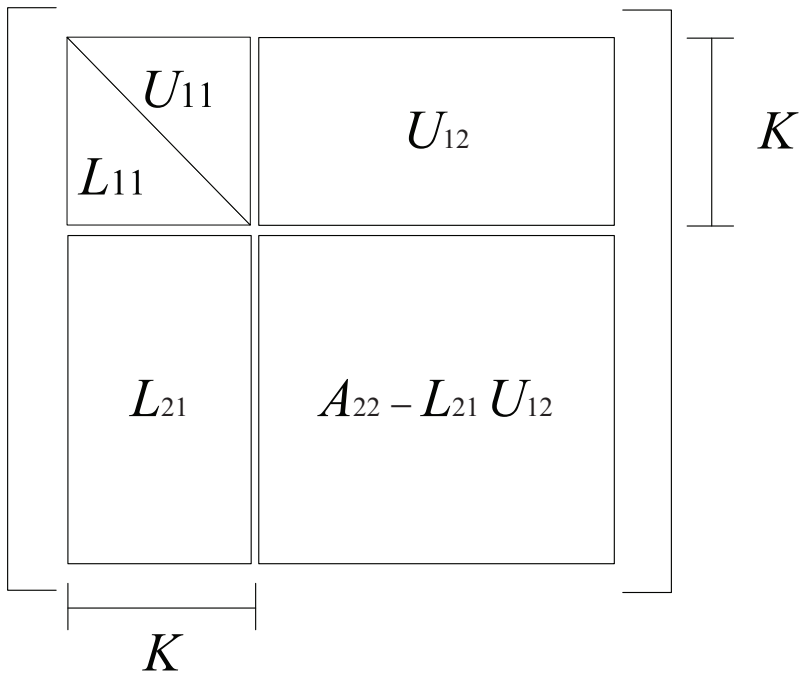


SPARK3 リダクション + DP-MP 混合精度反復改良法（右・黒棒）が最高速

Strassen アルゴリズムによる LU 分解の高速化

係数行列を LU 分解する ($A = LU$) 際に，下記のように行列積を使用する。

1. A を $A_{11} \in \mathbb{R}^{K \times K}$，$A_{12} \in \mathbb{R}^{K \times (n-K)}$, $A_{21} \in \mathbb{R}^{(n-K) \times K}$, and $A_{22} \in \mathbb{R}^{(n-K) \times (n-K)}$ に分割
2. A_{11} を $L_{11}U_{11} (= A_{11})$ に LU 分解し，A_{12} を U_{12}，A_{21} を L_{21} に変換
3. $A_{22}^{(1)} := A_{22} - \underline{L_{21}U_{12}}$
4. $A := A_{22}^{(1)}$ とし，$n - K \geq 0$ が成立しなくなるまで繰り返し



Winograd's variant による行列積計算 (1/2)

$$C := AB = [c_{ij}]$$

ここで，$C \in \mathbb{R}^{m\times n}$, $A = [a_{ij}] \in \mathbb{R}^{m\times l}$，$B = [b_{ij}] \in \mathbb{R}^{l\times n}$ とする。

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},\ B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}. \tag{11}$$

$$\begin{aligned} S_1 &:= A_{21} + A_{22},\ S_2 := S_1 - A_{11}, \\ S_3 &:= A_{11} - A_{21},\ S_4 := A_{12} - S_2, \\ S_5 &:= B_{12} - B_{11},\ S_6 := B_{22} - S_5, \\ S_7 &:= B_{22} - B_{12},\ S_8 := S_6 - B_{21}, \end{aligned} \tag{12}$$

$$\begin{aligned} &M_1 := S_2S_6,\ M_2 := A_{11}B_{11}, M_3 := A_{12}B_{21}, \\ &M_4 := S_3S_7,\ M_5 := S_1S_5,\ M_6 := S_4B_{22}, \\ &M_7 := A_{22}S_8, \end{aligned} \tag{13}$$

Winograd's variant による行列積計算 (2/2)

$$T_1 := M_1 + M_2,\ T_2 := T_1 + M_4. \tag{14}$$

Through (12)$\rightarrow$ (13) $\rightarrow$ (14), we can obtain C as follows:

$$C := \begin{bmatrix} M_2 + M_3 & T_1 + M_5 + M_6 \\ T_2 - M_7 & T_2 + M_5 \end{bmatrix}$$

Winograd's variant involves the following arithmetical operations:

$$\begin{aligned} \mathsf{Mul}(m,l,n) &= 7\mathsf{Mul}(m/2,l/2,n/2), \\ \mathsf{Addsub}(m,l,n) &= 4\mathsf{Addsub}(m/2,l/2) \\ &+ 4\mathsf{Addsub}(l/2,n/2) \\ &+ 7\mathsf{Addsub}(m/2,n/2). \end{aligned}$$

ベンチマーク

Table: Computation time: Strassen's and Winograd's algorithms (128 bits)

$m \times n$	min(Simple, Block)	Strassen	Winograd
255 × 255	1.06	0.72	0.63
256 × 256	1.22	0.70	0.57
257 × 257	1.04	0.74	0.60
511 × 511	9.60	4.84	4.06
512 × 512	9.68	4.77	3.73
513 × 513	9.89	4.92	3.88
1023 × 1023	77.63	32.02	25.57
1024 × 1024	77.72	31.53	24.10
1025 × 1025	78.41	32.21	24.77
2047 × 2047	619.21	211.80	163.87
2048 × 2048	618.87	211.19	155.67
2049 × 2049	623.24	212.79	157.52

ベンチマーク

Table: Computation time: Strassen's and Winograd's algorithms (1024 bits)

$n \times n$	min(Simple, Block)	Strassen	Winograd
255 × 255	5.46	2.33	1.95
256 × 256	5.53	2.31	1.72
257 × 257	5.61	2.41	1.81
511 × 511	43.81	13.40	10.57
512 × 512	44.20	13.02	9.44
513 × 513	44.37	13.38	9.81
1023 × 1023	352.79	76.93	57.98
1024 × 1024	355.99	74.58	52.47
1025 × 1025	356.58	76.36	54.22
2047 × 2047	2820.16	454.02	329.41
2048 × 2048	2824.34	446.87	302.56
2049 × 2049	2829.95	456.08	307.05

悪条件の連立一次方程式

Lotkin 行列 $a_{ij} = \begin{cases} 1 & (i = 1) \\ 1/(i+j-1) & (i \geq 2) \end{cases}$

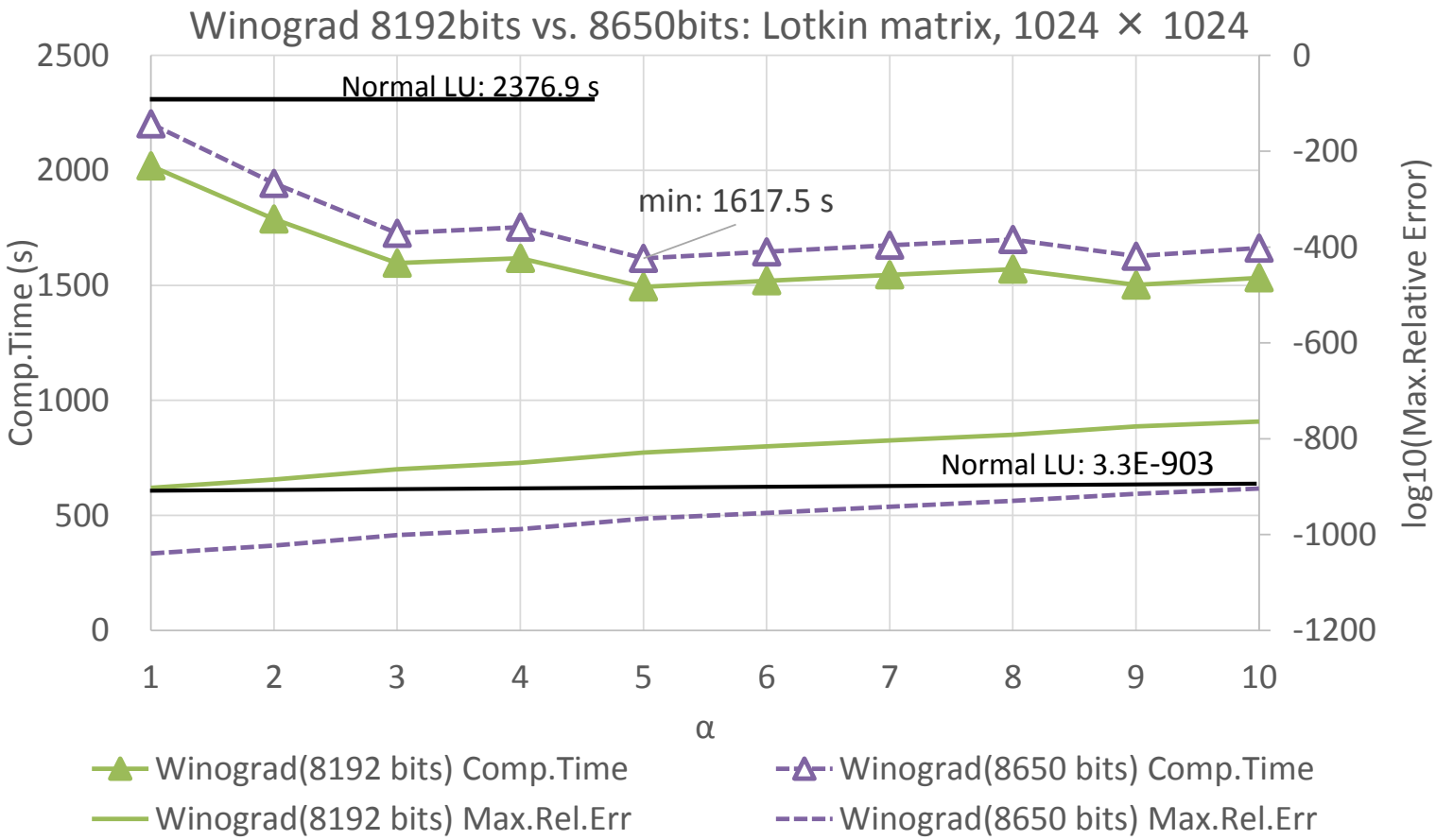
条件数 $\|A\|_1 \|A^{-1}\|_1 \approx 4.3 \times 10^{1576}$ $(n = 1024)$

真の解 $\mathbf{x} = [0\ 1\ \ldots\ n-1]^T$

α $K := 32\alpha$ として最短の計算時間と相対誤差をチェック

Lotkin 行列の場合

- 相対誤差は 10 進約 138 桁増大する ($n = 1024$ の場合)
- 8650bits 計算 (458bits $\approx$ 10 進 137.87 桁) を行って相対誤差の増大分をカバー



→ 32 %の計算時間削減が可能

- 自己紹介
- 多倍長数値計算＝多倍長浮動小数点演算の基本
- 多倍長数値計算の高速化
- 連立一次方程式の解計算の高速化
- まとめと今後の課題

まとめ

- 1CPU の高速化はほぼ限界に達している→マルチコア・メニーコア環境を利用した並列化による高速化が必須
- 連立一次方程式（＋線型計算）の高速化がカギ（と考えている）
- 問題の精度に応じた高速化手法を使い分ける必要がある

今後の課題

1. GPU を用いた高速化の追求
2. 陰的解法の追求
3. 関数近似解法の追求　→　多倍長関数計算の高速化