2004－ARC－159　(12)
2004／7／30

命令キャッシュを考慮したコード生成法による方程式求解の高速化手法

根本和宜[†]　佐田宏史[†]　前川仁孝[†]
伊與田光宏[†]　宮崎収兄[†]

本稿では，スパース行列を係数行列に持つ連立一次方程式の高速求解手法であるコード生成法において，命令キャッシュを有効利用することで高速化する手法を提案する．従来のコード生成法は，繰り返し演算のない算術代入文を列挙したループフリーコードを生成する．このため，演算を実行する毎にメモリから命令をフェッチする必要があり，高速実行の妨げとなる．そこで，CISC 命令ライクの命令コードを生成し，これを命令キャッシュサイズより少ない量の命令で繰り返し解釈しながら実行する．最後に，提案する手法の有効性を汎用プロセッサ Pentium4 上で評価した．評価の結果，従来より最大 2.53 倍の速度向上が確認された．

The High Speed Scheme of Simultaneous Equations Solution by the Code Generation Method to Improve Instruction Cache

Kazuyoshi Nemoto ,[†] Hiroshi Sata ,[†] Yoshitaka Maekawa ,[†]
Mituhiro Iyoda [†] and Yoshinobu Miyazaki[†]

This paper proposes a high speed scheme by the code generation method with using the instruction cache effectively. The conventional code generation method generates a loop free code which is enumerated arithmetic assignment statement without loop operation. For this reason, when operations are executed, it is necessary to fetch instructions from memory. It becomes the hindrance of speedup. Therefore, the proposed method generates the intermediate code similar to CISC instruction. The intermediate code is decoded and executed repeatedly. Hence, the intermediate code is able to fetch instructions from the instruction cache. The effectivity of proposed method is evaluated on the general processor Pentium4. The result of evaluation is that the proposed scheme is obtained up to 2.53 times speedup compared with a loop free code.

1. はじめに

数値計算の分野では，連立一次方程式求解を必要とする問題が多くある．しかし，連立一次方程式求解の処理時間は，問題の大規模化などにより増加する一方であり，処理時間の短縮が望まれている．連立一次方程式求解を必要とする問題の多くは，係数行列にゼロ要素を多く含むスパース行列であることが知られている[1]．連立一次方程式の求解において，係数行列がスパース行列の場合，クラウト法などの一般的な解法では，ゼロ要素を含む四則演算などの無駄な演算が多くの処理時間を占める．

係数行列がスパース行列である連立一次方程式を高速に求解する手法の一つにコード生成法[2] がある．コード生成法は，方程式求解に必要な演算のみを直接実行可能なコードとして生成する手法である．コード生成法により生成される実行コードは，繰り返し演算のない算術代入文を列挙したループフリーコードとなる．コード生成法は一般的な解法より数十倍高速[3][4] であることが知られている．

問題の大規模化に伴い，コード生成法の更なる高速化が望まれている．このため，生成するコードのデータ依存関係を解析して実行コードを並列処理する手法[5][6] や，PC と専用ハードウェアを用いて協調処理する手法[7] などが提案されている．しかし，これらは専用のハードウェアを必要とするため，近年では汎用プロセッサを用いたコード生成法の研究が多く行われている．例として，特定の OS 上でコード生成法を実装する手法[8] や，スカラ変数を用いて配列間接アクセ

† 千葉工業大学 情報工学科
Department of Computer Science, Chiba Institute of Technology

スをなくし高速に処理する手法[9] などが提案されている．しかし，汎用プロセッサ上でコード生成法を用いて求解する場合，生成されるループフリーコードは演算する度にメモリから命令をフェッチする必要がある．このため，現在多くの汎用プロセッサが持つ命令キャッシュを有効利用することができない．これは，命令キャッシュとしてトレースキャッシュを持つ汎用プロセッサである Pentium4 において更に顕著化する．

そこで本稿では，コード生成法により生成されるコードとして，実行コードとは異なる中間コードを生成し，連立方程式求解の際に使用する演算命令を命令キャッシュからフェッチすることで，命令キャッシュを有効利用するコード生成法を提案する．以下本稿では，2 章でコード生成法ついて述べ，3 章で汎用プロセッサにおけるコード生成法及びその問題点について述べ，4 章で問題点を解決するためのコード生成法の提案を行う．最後に，5 章で提案する手法の有効性を評価し，6 章で全体をまとめる．

2. コード生成法

クラウト法を用いて連立一次方程式 $A\boldsymbol{x}=\boldsymbol{b}$ を求解する場合，係数行列 A を $A=LU$ で表される下三角行列 L と上三角行列 U に分解し，前進代入，後退代入することで解を求める．クラウト法による連立方程式の求解は，係数行列のサイズを N，係数行列 A の各要素を a_{ij}，解行列 $\boldsymbol{b}$ の各要素を b_i とすると，図 1 のような計算により解が b_i に求まる．図中の 1 行目から 13 行目までが LU 分解，14 行目から 19 行目までが前進代入，20 行目から 24 行目までが後退代入である．これらの LU 分解及び前進代入，後退代入に必要な演算は，以下のような式 (1)，式 (2) のみで構成されている．

$$op1 = op1 - op2 \times op3 \quad (1)$$

$$op1 = op1/op2 \quad (2)$$

コード生成法は，式 (1)，式 (2) からなる LU 分解及び前進代入，後退代入の演算のうち，ゼロ要素を含む四則演算のような無駄な演算を除いた実行コードを生成し，その実行コードを直接実行することで解を得る手法である．コード生成法により生成される実行コードは，繰り返し演算のない算術代入文を列挙したループフリーコードとなる．このループフリーコードは，データ構造が変化しない限り毎回生成する必要がなく，係数行列の値だけ変えて繰り返し計算できることから，電子回路シミュレーションで扱うような非ゼロ要素の値だけが変化し，データ構造がほとんど変化しないような処理において特に効果的である[4)5)7)9)]．

```
1:    for i = 1 to N
2:        for j = i to N
3:            for k = 1 to i − 1
4:                a_ji = a_ji − a_jk × a_ki
5:            end
6:        end
7:        for j = i + 1 to N
8:            for k = 1 to i − 1
9:                a_ij = a_ij − a_ik × a_kj
10:           end
11:           a_ij = a_ij / a_ii
12:       end
13:   end
14:   for i = 1 to N
15:       for j = 1 to N
16:           b_i = b_i − a_ij × b_j
17:       end
18:       b_i = b_i / a_ii
19:   end
20:   for i = N to 1
21:       for j = i + 1 to N
22:           b_i = b_i − a_ij × b_j
23:       end
24:   end
```

図 1 クラウト法による連立方程式求解
Fig. 1 An Equations Solution by Crout Method

3. 汎用プロセッサにおけるコード生成法とその問題点

近年，コード生成法は急速に処理性能が向上した汎用プロセッサへの実装[8)] が主に行われている．代表的な汎用プロセッサの一つに intel 社の Pentium4 がある．Pentium4 は，命令キャッシュにトレースキャッシュをもつ．トレースキャッシュは，一度実行した命令を実行した順番通りにキャッシュに格納する．Pentium4 のトレースキャッシュは，デコーダより後に配置されていて，CISC 命令を格納するのではなく，デコードされたマイクロオペレーションと呼ばれる RISC 命令を格納する[10)]．これにより，頻繁に実行される命令に対し，命令のフェッチとデコードに要する時間が削減される．

コード生成法により生成されるループフリーコードは，算術代入文を列挙したループ文のないコードであるため，実行に必要な演算命令は一命令につき一

表 1 演算式と中間コード
Table 1 An Arithmetic Expression and the Intermediate Code

演算式	オペレータ	オペランド
$element[x] = element[x] - element[y] * element[z];$	1	x, y, z
$element[x] = element[x]/element[y];$	2	x, y
$element[x] = element[x] - element[y_1] * element[z_1];$ $\vdots$ $element[x] = element[x] - element[y_n] * element[z_n];$	3	$n, x, y_1, z_1, \ldots, y_n, z_n$
—	-1	—

度しか実行されない．このため，Pentium4 上でループフリーコードを実行した場合，演算命令をトレースキャッシュからフェッチすることができない．また，Pentium4 におけるメモリからフェッチする演算命令は CISC 命令であるため，実行する度に毎回デコード処理を行う必要がある．つまり，Pentium4 上でループフリーコードを実行した場合，トレースキャッシュの有効利用による処理の高速化ができないだけでなく，メモリから演算命令をフェッチする度にデコード処理する時間が必要になる．

4. トレースキャッシュを考慮したコード生成法

コード生成法における命令フェッチ及びデコードにかかる時間を改善するためには，常にトレースキャッシュから演算命令をフェッチする必要がある．そこで，本稿では演算に必要な情報を格納した CISC 命令ライクな中間コードを生成する．この中間コードは，実行する演算式と演算する要素を決定する情報からなり，トレースキャッシュに収まる程度の小さなプログラムにより解釈しながら実行される．ここで，中間コードを生成する部分を中間コード生成部，生成された中間コードを解釈しながら演算する部分を中間コード実行部とする．以下に，これらの詳細について述べる．

4.1 中間コード生成部

中間コード生成部では，LU 分解及び前進代入，後退代入を行う演算式から CISC 命令ライクな中間コードを生成する．生成する中間コードの各命令はオペレータと複数のオペランドから構成される．表 1 に，コード生成法で実行する演算式と対応する中間コードを示す．中間コードは基本的にオペレータ 1 及びオペレータ 2 で構成される．オペレータ 1 は式 (1) に対応した算術代入演算で，オペランドは式 (1) の演算に必要な 1 つの更新要素の情報と 2 つの参照要素の情報からなる．また，オペレータ 2 は式 (2) に対応した算術代入演算で，オペランドは式 (2) の演算に必要な 1 つの更新要素の情報と 1 つの参照要素の情報からなる．

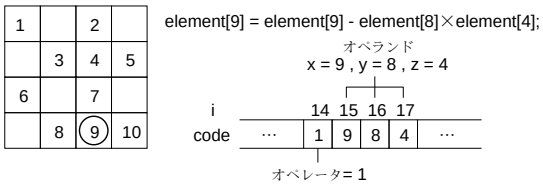

図 2 中間コード生成例
Fig. 2 An Example of the Intermediate Code Generation

オペレータ 3 は，LU 分解における三重ループ中の最内ループが，同一要素を同一の演算式 (式 (1)) により，値を繰り返し更新する場合に生成するオペレータである．オペレータ 3 に対応する先頭のオペランドは，演算の繰り返し回数が格納される．詳細は次節で述べるが，中間コード実行部の処理を最適化するために繰り返しの最初と最後は別途処理する必要があるので，繰り返し回数として格納される値は，実際に演算を繰り返す回数より 2 少ない数となる．2 番目のオペランドは，値の更新対象の要素の位置を示す情報が格納される．3 番目以降のオペランドは，参照対象の要素の位置を示す情報の組が連続して格納される．演算を繰り返す回数を n とするとオペランドの総数は $2n + 2$ 個となる．

オペレータ-1 は中間コードの最後を示すオペレータである．このオペレータは中間コードの最後に 1 つだけ存在する．

図 2 に中間コードの生成例を示す．図 2 中の左側の行列は，連立方程式を求解する際の係数行列を，右側は生成したコードの例を示す．行列内の数字は LU 分解中に発生する fill-in を含めた非零要素の位置を表している．非零要素は配列 *element* に格納され，図中に示す数字でアクセスできるものとする．図 2 において，要素 $element[9]$ をコード生成法を用いて LU 分解するには $element[9] = element[9] - element[8] \times element[4]$ という演算を行う．既に，$element[7]$ まで LU 分解が終了していて $code[13]$ まで中間コードが生成されているとすると，$element[9]$ の値を更新するために

使用する演算式のオペレータは，表 1 より 1 であるので $code[14]$ には 1 が格納される．また，これに続くオペランドは，式 (1) より 3 つであり，表 1 から $x = 9$，$y = 8$，$z = 4$ となる．このため，$code[15]$ に格納されるコードは 9，$code[16]$ に格納されるコードは 8，$code[17]$ に格納されるコードは 4 となる．このように LU 分解を行うための全ての中間コードを演算順に生成する．全ての中間コードを生成したら，$code$ の最後に終了フラグを示すオペレータ-1 を格納し，中間コードの生成を終了する．

4.2 中間コード実行部

中間コード実行部は，図 3 のように，各オペレータを解釈しながら実行する処理部分を内部に持つループ構造である．中間コード実行部では，まず中間コードからオペレータを読み込み，表 1 に従って実行する演算式を決定する．次にオペランドの読み込みを行う．オペレータが 1 である場合はオペランドを 3 つ，オペレータが 2 である場合はオペランドを 2 つ読み込む．オペレータ 1 及びオペレータ 2 のオペランドは，演算式が必要とする要素へアクセスするためのインデックスとして参照し，算術代入演算を実行する．またオペレータが 3 である場合は，まず繰り返し回数となるオペランドを 1 つ読み込む．この値は，繰り返しの最初のロードと最後のストアを別途処理するため，実際に連続する算術代入演算の数より 2 少ない数になる．次にオペランドを 3 つ読み込み，オペレータ 1 と同様にオペランドの値をインデックスとして使用する算術代入演算を実行する．次に繰り返し回数だけオペランドを 2 つずつ読み込み，対応する算術代入演算の実行を繰り返す．繰り返し演算中の値の更新はレジスタ上で行うので高速に処理ができる．最後に，オペランドを 2 つ読み込み，対応する算術代入演算を実行する．またオペレータが-1 である場合は実行を終了する．

図 4 に中間コードを解釈し実行する例を示す．ここで，中間コードは $code[13]$ まで解釈及び実行を終えているとする．中間コード実行部は，まず中間コード $code[14]$ を読み込む．オペレータは $code[14] = 1$ であるので，表 1 より式 (1) を演算するオペレータと解釈する．次に，表 1 よりオペランドを 3 つ読み込み，値をレジスタにロードする．レジスタにロードする値は，$code[15] = 9$，$code[16] = 8$，$code[17] = 4$ となり，演算要素へのインデックス値として参照され，$element[9] = element[9] - element[8] \times element[4]$ の算術代入演算を実行する．演算結果の値で要素 $element[9]$ を更新する．

```
while
    if(code[i] = 1)
        code[i+1]からr1に値をロード
        code[i+2]からr2に値をロード
        code[i+3]からr3に値をロード
        r4 = element[r3] × element[r2]
        r4 = element[r1] - r4
        element[r1] = r4
        i = i + 4
    else if(code[i] = 2)
        code[i+1]からr1に値をロード
        code[i+2]からr2に値をロード
        r3 = element[r2] / element[r1]
        element[r2] = r3
        i = i + 3

    else if(code[i] = 3)
        code[i+1]からr1に値をロード
        code[i+2]からr2に値をロード
        code[i+3]からr3に値をロード
        code[i+4]からr4に値をロード
        r5 = element[r4] ×element[r3]
        r5 = element[r2] - r5
        i = i + 5
        for( j = 0 ; j < r1 ; j++)
            code[i]からr3に値をロード
            code[i+1]からr4に値をロード
            r6 = element[r4] ×element[r3]
            r5 = r5 - r6
            i = i + 2
        end
        code[i]からr3に値をロード
        code[i+1]からr4に値をロード
        r6 = element[r4] × element[r3]
        r5 = r5 - r6
        element[r2] = r5
        i = i + 2
    else if(code[i] = -1)
        演算終了
end
```

図 3 中間コード実行部の構造
Fig. 3 Structure of the Intermediate Code Execution

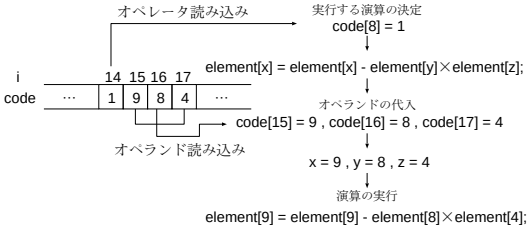

図 4 中間コードの解釈及び実行例
Fig. 4 An Example of the Intermediate Code of Decoding and Execution

5. 性能評価

中間コード実行部において，オペレータ 3 は繰り返し処理により値を更新する際，不要なデータのロードとストアをなくす最適化を行っているので高速化が期待できる．しかしそのオーバヘッドが発生するので，オペレータ 1 と比べて遅くなる可能性がある．そこで，以下の節では，オペレータ 1 による処理よりも，オペレータ 3 による処理が高速化する処理の繰り返し回数を求め，それを用いて，コード生成法により生成した中間コードを用いて提案手法の有効性を評価する．

評価は，CPU に Pentium4 3.06GHz を持つ計算機上で行う．OS は Linux 2.4.26，メモリは 2GBytes である．使用した Pentium4 は，トレースキャッシュ 12Kμops，L1 データキャッシュ8KBytes，L2 キャッシュ512KBytes であり，キャッシュラインサイズは L1 キャッシュ，L2 キャッシュともに 64Bytes である．なお，時間の計測は gettimeofday の関数を用いて，マイクロ秒単位で測定した．

5.1 オペレータ 1 とオペレータ 3 のコストの比較

更新操作の連続回数について検討する．評価には，オペレータ 1 を 120000 回連続して実行した場合の処理時間と，オペレータ 3 の最初のオペランドであるループ回数を 0 (実際の繰り返し回数は 2) の場合と，1 (実際の繰り返し回数は 3) の場合を比較した．この際，オペレータ 3 を連続させる回数は，繰り返し回数が 2 の場合，オペレータ 1 の半分の回数 (60000 回)，繰り返し回数が 3 の場合，オペレータ 1 の 1/3 回 (40000 回) となる．評価の結果，オペレータ 1 を連続して実行した場合の処理時間は 1222[μsec]，オペレータ 3 におけるループ回数を 0 とした場合の処理時間は 801[μsec]，ループ回数を 1 とした場合の処理時間は 678[μsec] となった．これらの結果からループ回数が 0 の場合，既にロード/ストア命令のコストがループオーバヘッドを上回っていることがわかる．つまり，同一要素を更新する演算式が 2 つ以上続く場合，オペレータ 3 を生成した方が良いといえる．

5.2 ループフリーコードとの処理時間における比較

提案する手法の有効性を示すため，提案手法により生成した中間コードを解釈しながら実行した処理時間と，アセンブラで生成したループフリーコードの処理時間を比較する．評価の際に使用したループフリーコードは，同一要素を繰り返し更新する処理には，コード生成法と同様の手法でレジスタを最適化したものを利用する[3]．評価に使用するデータは，行列サイズとスパース率を指定し，MT 法[11] を用いてランダムに要素を発生させたあと，fill-in 数を削減するため Markowitz の方法[12] を用いてオーダリングを行い生成した．

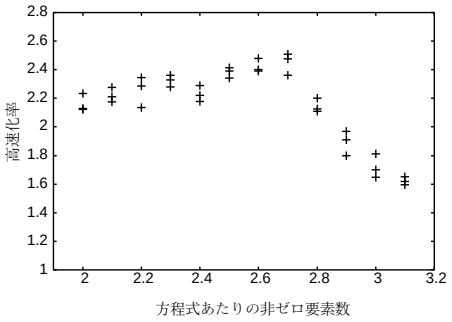

図 5 行列サイズ 2000 における非ゼロ要素数と高速化率の関係
Fig. 5 Relation Between the Number of Non-zero Elements in the Matrix Size 2000 and Speedup

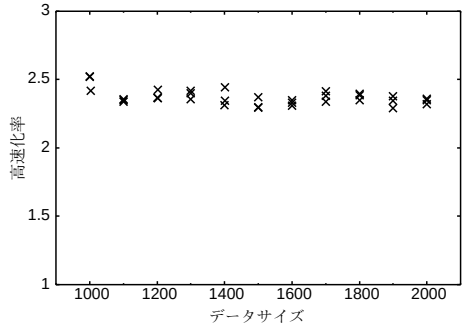

図 6 行列サイズと高速化率の関係
Fig. 6 Relation Between Matrix Size and Speedup

まず，連立一次方程式求解における係数行列の非ゼロ要素数が変化した場合の提案手法の高速化率について評価する．評価に用いたデータは，行列サイズは 2000 で，方程式あたりの非ゼロ要素数は 2.0 から 3.1 まで 0.1 刻みで各 3 例づつ計 36 例である．図 5 にループフリーコードの実行時間を基準としたときの提案するコード生成法の高速化率を示す．評価の結果，方程式あたりの非ゼロ要素数が少ない場合では，2 倍以上の高速化が得られているが，非ゼロ要素数が多い場合では，提案手法の従来手法に対する高速化率が低下していることがわかる． 方程式あたりの非ゼロ要素数が多いときは，fill-in により演算が増加する．よって提案手法では，fill-in の数を減らす効果的なオーダリングをすることで高速化を得ることができると考えられる．

次に，係数行列のサイズが変化した場合の高速化率

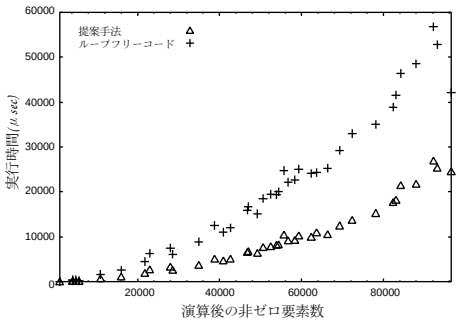

図 7 実行時間と演算後の非ゼロ要素数
Fig. 7 The Number of Non-zero Elements After Execution Time and Operation

について評価する．使用したデータは，データサイズが 1000 から 2000 まで 100 刻みで各 3 例づつの計 33 例であり，各データの方程式あたりの平均要素数は 2.5 個である．図 6 に方程式あたりの要素数が一定である異なるサイズのデータを用いた際の，提案するコード生成法の高速化率を示す．評価の結果，提案するコード生成法では方程式あたりの平均要素数が一定の場合，係数行列のサイズに関わらずループフリーコードより 2.5 倍程度高速に処理できたことが確認できた．よって，提案手法では方程式あたりの非ゼロ要素数の割合が一定であれば，ループフリーコードに対しての高速化率は行列サイズとは無関係であることがわかる．

最後に，LU 分解演算後の非ゼロ要素数における実行時間について評価する．使用したデータは演算後の非ゼロ要素数が均等になるようにスパース率を指定し，行列サイズが 1000，1500，2000，2500，3000 の中からランダムに生成した各 40 個のデータを使用した．図 7 に異なる演算量のデータを用いた際のループフリーコードと，提案するコード生成法の実行時間を示す．提案するコード生成法の実行時間は，ループフリーコードの実行時間と比較して，演算後の非ゼロ要素数に関係なく全体的に 2 倍前後高速であることが確認できた．また，演算後の非ゼロ要素数を n とすると，実行時間のオーダは $O(n^2)$ で表せる．よって，提案手法によりスパース連立方程式の求解を行う場合，演算後の非ゼロ要素数を削減すること，つまり fill-in の発生回数を削減するオーダリングをすることが重要である．

6. おわりに

本稿では，コード生成法において CISC 命令ライクの中間コードを生成し，これを Pentium4 のトレースキャッシュ上で解釈及び実行することで，命令キャッシュを有効に利用する手法を提案した．評価の結果，スパース率によって高速化率は変動するものの，ループフリーコードに対し 2 倍程度の高速化が確認できた．今後は，本提案手法を電子回路シミュレータ SPICE に実装し評価する予定である．

参考文献

1) 小国 力: 行列計算ソフトウェア, 丸善, (1991).
2) F.G.Gustavson, W.Liniger, R.Willoughby: Symbolic Generation of an Optimal Crout Algorithm for Sparse Systems of Linear Equations, Journal of the Association for Computing Machinery, Vol. 17, No. 1, pp. 87-109, (1970).
3) 福井 義成: 問題の性質を利用した大規模スパース行列の高速解法, 情報処理学会研究報告「数値解析」, No. 20-3, (1987).
4) Y. Fukui, H. Yoshida, S. Higono: Supercomputing of Circuit Simulation, Proc. Supercomputing '89, pp. 81-85, (1989).
5) 前川 仁孝, 高井 峰生, 伊藤 泰樹, 西川 健, 笠原 博徳: スタティックスケジューリングを用いた電子回路シミュレーションの粗粒度/近細粒度階層型並列処理手法, 情報処理学会論文誌, Vol. 37, No. 10, (1996).
6) H. Kasahara, W. Premchaiswadi, M. Tamura, Y. Maekawa, S. Narita: Parallel Processing of Sparse Matrix Solution Using Fine Grain Tasks on OSCAR (Optimally Scheduled Advanced Multiprocessor), International Conference on Parallel Processing, (1991).
7) 八木 浩行, 檀 良: LUCAS を使用した回路シミュレータの性能評価, 情報処理学会論文誌, Vol. 41, No. 4, pp. 915-926, (2000).
8) 八木 浩行, 檀 良: PC 環境での回路シミュレーション用実行時コード生成法, 電子情報通信学会論文誌 (A), Vol. J83-A, No. 4, pp. 444-446, (2000).
9) 間中 邦之, 刑部 亮, 前川 仁孝, 笠原 博徳: 配列間接アクセスを用いないコード生成法による電子回路シミュレーションの高速化とその並列処理, 情報処理学会 ARC137-14/HPC80-14, (2000).
10) Intel Corporation: IA-32 Intel Architecture Optimization, http://developer.intel.com/ design/Pentium4/documentation.htm
11) M. Matsumoto, T. Nishimura: Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator, ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp. 3-30, (1998).
12) H. M. Markowitz: The Elimination Form of Inverse and Its Application to Linear Programming, Management Science, Vol. 3, pp. 255-269, (1957).