計算機アーキテクチャ　147−27
ハイパフォーマンス　89−27
コンピューティング
（2002．3．8）

ブロック幅を動的決定する疎行列連立一次方程式の直接解法

西出隆二[†]　片桐孝洋[††]　金田康正[†††]

LU 分解の一手法であるスーパーノードアルゴリズムに，ブロック幅を最適化する自動チューニング機構を採用した．5 種の異なる計算機でその効果を評価したところ，最大で 1.5 倍程度の性能差が認められた．このアルゴリズムはブロック化された内積形式ガウス法の一種であるが，ブロック幅の最適値を自動探索する機構は利用者の負担を軽減し，求解までの時間を短縮するために必要不可欠である．また最適値として選択された値はアーキテクチャによらず行列の構造を反映していることがわかった．これは係数行列の非零要素の位置を定量的にモデル化することで，探索時間と求解時間のさらなる短縮が可能であることを示唆している．したがって定量的モデル化が自動チューニングには必須である．

Direct Solver for Sparse Linear Equations with Dynamic Block Size Selection

Ryuji Nishide ,[†] Takahiro KATAGIRI [††]
and Yasumasa KANADA[†††]

We adopted the auto-tuning mechanism of optimizing the blocking parameter automatically to the supernodal algorithm (one of LU decomposition methods). Using five different architectures, we found this auto-tuning mechanism achieved at most 1.5 times faster than the not tuned case. It shows an important finding that the user's load and solving time are reduced in the "blocked" left-looking algorithm. We also found that the optimized parameter related to the structure of solving matrices. This implies that we could reduce the searching and solving time by modeling the non-zero structure of the coefficient matrix quantitatively.

1. はじめに

係数行列が疎行列となる連立一次方程式は構造解析等で良く見られ，反復解法で求解が困難な場合には *LU* 分解を用いた直接解法が多用される [1]．直接解法は行列構造やアーキテクチャ，コンパイラ等に応じて最適なアルゴリズムやパラメータを選択する必要がある．しかし，従来の BLAS，LAPACK，LINPACK，BLACS，ScaLAPACK 等の数値計算ライブラリはこれをユーザの判断に委ねており，利用上の大きな負担となっていた [2]．

そこで ILIB を始めとする自動チューニング機構を付加した数値計算ライブラリが開発されている [3]．自動チューニングとは求解に最も適したアルゴリズムやパラメータを，利用者のサポートを必要とせず全て自動で選択する仕組みである．これは，行列構造等に応じたチューニングが演算時間を大きく削減できること，またその作業を利用者が行うのは負担増となることの二点から，意義深い機構であると考えられる．

そこで本論文では，近年疎行列直接解法で注目をあびているスーパーノードアルゴリズム [4] に自動チューニング機構を加え，ブロック幅等のパラメータを動的に変えて最適な値にチューニングする効果を検証した．スーパーノードアルゴリズムとはブロック化された left-looking アルゴリズムの一種であり，近年よく見られるようになった階層メモリ構造を持つアーキテクチャ上で高い性能を発揮するアルゴリズムである．本論文で提案する自動チューニング機構の特徴は，アーキテクチャの構造（キャッシュサイズ等）により，プログラム実行前に値を決定できるものと，行列構造により実行時に決定するものの二種類に分類し，両者を組み合わせる機構である．

本論文の構成は以下の通りである．まず 2 章で一般的なスパースソルバの流れを説明した後，3 章でスーパーノードアルゴリズムの説明，4 章で自動チューニングの手法について述べる．5 章では評価結果を示し，最後に 6 章でまとめと今後の課題を述べる．

† 東京大学大学院新領域創成科学研究科基盤情報学専攻
Department of Frontier Informatics, Graduate School of Frontier Sciences, the University of Tokyo
†† 科学技術振興事業団 さきがけ研究２１ (情報基盤と利用環境) 領域
"Information Infrastructure and Applications",PRESTO, Japan Science and Technology Corporation(JST)
††† 東京大学情報基盤センタースーパーコンピューティング研究部門
Computer Centre Division, Information Technology Center, the University of Tokyo

2. スパースソルバ

2.1 全体の流れ

スパースソルバの処理の流れは図 1 で表される．以下の処理を順次行うことで A が非対称の連立一次方程式 $Ax = b$, $(A \in \Re^{n*n}, x \in \Re^n, b \in \Re^n)$ の解を求める．

(1) オーダリング
(2) シンボリック分解
(3) LU 分解
(4) 前進後退代入

各処理について以降，簡単に説明する．

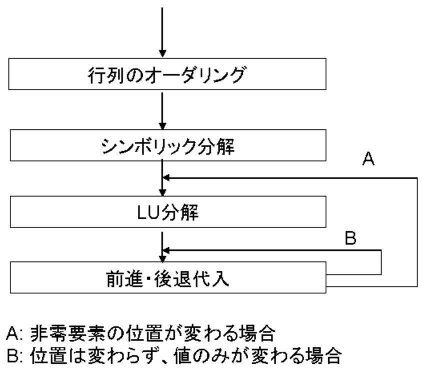

図 1：スパースソルバのフローチャート．

2.2 オーダリング

オーダリングとは行交換行列 P と列交換行列 Q を適切に定め，PAQ を LU 分解したときのフィルインができるだけ少なくなるようにする処理のことである．オーダリングの手法としては三角化（Markowitz の方法，Tewarson の方法等）や帯幅縮小（RCM 法，最小次数順序法等），ブロック化（Stewart の方法，dissection 分割法）等多くのアルゴリズムが知られている [5]．

本論文で紹介するスーパーノードアルゴリズムは LU 分解の各段階で部分ピボッティングを行なうため，ピボッティングにより行列の疎性が失われると懸念される．しかし，[6] によれば，L_c を A^TA のコレスキー分解後の上三角行列とすると，L が $P_1L_1P_2L_2...P_{n-1}L_{n-1}$ ☆のような形で格納されているならば，L の構造は L_c に含まれ，U の構造は L^T に含まれる．これは A の数値によらず成り立つので，A^TA に対して最適なオーダリング行列 Q を見つければ，それを A のオーダリング行列として使うことができる．

本論文で用いるスーパーノードアルゴリズムには，Matlab にも用いられオーダリングとして標準的な最小次数順序法 [7] が使われている．

2.3 シンボリック分解

シンボリック分解とは，行列 A を LU 分解したときの非零要素の位置を計算し，LU 分解に先立って必要なメモリ領域を確保するための処理である．原理的には LU 分解において，通常の演算の代わりに，要素の値が零か非零かのみに着目して演算を行うことで実現できる．しかし，この方法では演算量が LU 分解と同等となってしまう．そこで，シンボリック分解を効率良く行うために列消去木 [4] という概念が利用される．列消去木とは非対称行列 A から計算される A^TA をコレスキー分解した後の行列 L から得られる節点数$=N$ のグラフである．各節点は L の各列に対応する．2 つの節点 i と j に対し，式 (1) が成り立つとき，i を j の親であると定義する．

$$i = \min\{k > j | L_{kj} \neq 0\} \quad (1)$$

図 2 に列消去木の例を示す．[9] によれば，A の非零要素数のオーダで A から列消去木を求めることができる．

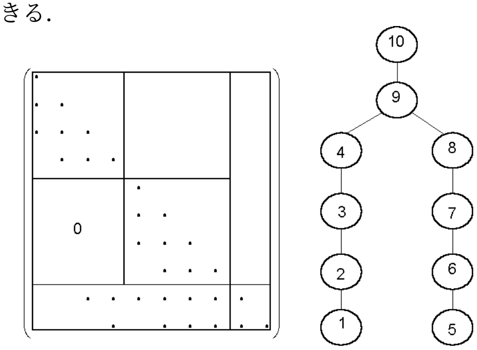

図 2：列消去木の例．

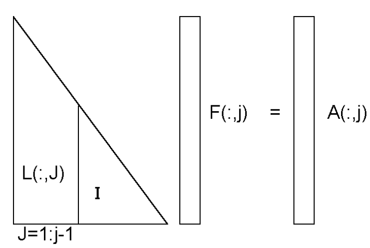

図 3：シンボリック分解のパラメータ．

列消去木を使ったシンボリック分解は次のようにして行う [10]．図 3 において，$i \xrightarrow{F} j$ は，有向グラフ F において i から j にエッジが張られているとする．これは F から作られる隣接行列の要素を f_{ij} としたときに，$f_{ij} \neq 0$ と同値である．$i \overset{F}{\Rightarrow} j$ は，F において i から j に direct path が張られているとする．また L を LU 分解後の下三角行列とする．[10] によれば，次の定理が成り立つ．

定理

「f_{ij} が非零であること（つまり，$i \xrightarrow{F} j$）と，ある $k (k \leq i)$ について $i \overset{L(:,J)}{\Rightarrow} k \xrightarrow{A} j$ が成り立つことは同値である.」

この定理を用いると，シンボリック分解は有向グラ

☆ $P_1, P_2, ..., P_{n-1}$ はピボッティングのための行交換行列

フ A と L の経路探索の問題に帰着することができる．

2.4 LU 分解

シンボリック分解によって確保されたメモリ領域をもとに，実際に LU 分解を行う．LU 分解の主な計算方法として，更新される列の右側が参照される right-looking アルゴリズム，左側が参照される left-looking アルゴリズム，上側と左側が参照される crout アルゴリズムが知られている [11]．なお，本論で取り扱うスーパーノードアルゴリズムは left-looking アルゴリズムを基盤にしている．

2.5 前進後退代入

LU 分解後の下三角行列 L を使った前進代入，上三角行列 U を使った後退代入から解 x を求める．もし，係数行列が同じ非零構造と非零値を持つ方程式を繰り返し解くのであれば，前進後退代入のみを繰り返せばよい．また，係数行列が同じ非零構造を持つ方程式の場合は，LU 分解に戻って処理を反復することになる．

3. スーパーノードアルゴリズム

3.1 スーパーノードとは

スーパーノードとは**「L において上部三角領域が全て非零で，各列が同じ非零構造を持った列の集合」**である．図 4 にスーパーノードの例を載せた．left-looking アルゴリズムにスーパーノードを導入することで，ブロック化が促進され，キャッシュにおけるデータの再利用性が増す．また，演算の中心となる行列ベクトル積を密行列のようなデータ参照で行うことができる (図 4)．

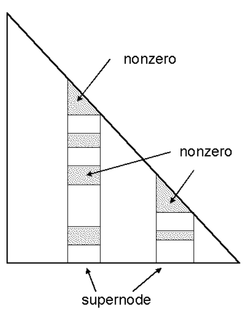

図 4：スーパーノードの例．

3.2 スーパーノードの結合

ブロック化に効果的なスーパーノードを得るために，スーパーノード生成の条件を緩和して，複数のスーパーノードを結合する手法が知られている．次のような結合規則を設ける [8]．

「消去木において子節点側の木が高々 r 個の節点を持つ場合に，部分木同士を融合する．」

本論文の LU 分解には，適当な r により融合されたスーパーノードを用いることにする．

3.3 スーパーノードを使った LU 分解

非対称行列に対する，スーパーノードを使った LU 分解は図 5，図 6 のように表される．

スーパーノードによって更新される列群を「パネル」と呼ぶことにする．灰色に塗られた領域同士のベクトル行列積が演算の中心を占めることになる．

まずシンボリック分解を行い，fill-in に相当する非零要素の位置を計算する．次にパネル左側の全てのスーパーノードが，順にパネル内の列群を更新する．更新は left-looking アルゴリズムを用いて行われる．最後に，更新されたパネル内で LU 分解を行うことで全体の LU 分解が完了する．

なお，スーパーノード内において行群をブロック化し，行ブロック単位で更新演算を行う方法が提案されている [4]．これは，スーパーノードのサイズがキャッシュに比べて大きく，キャッシュミスが多発するようなケースに有効である．ここで，行ブロックのサイズを b とする．この b は後述の自動チューニング機構に用いるパラメータの一つである．

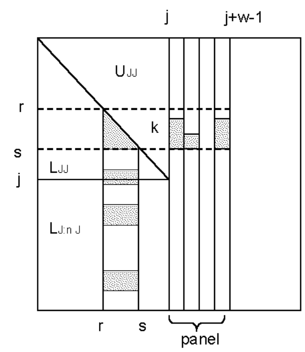

図 5：スーパーノードアルゴリズムを使った LU 分解．

for column j=1 **to** n **step** w **do**
  Symbolic factorization;
  for each updating supernode $(r:s) < j$ in topological order **do**
    for column $jj = j$ **to** $j + w - 1$ **do**
      $f = A(:, jj)$;
      $f(r:s) = L(r:s, r:s)^{-1} \cdot f(r:s)$;
      $f(s+1:n) = f(s+1:n) - L(s+1:n, r:s) \cdot f(r:s)$;
    end for;
  end for;
  Inner factorization: Apply the LU decomposition within the panel
end for;

図 6：スーパーノードアルゴリズム．

4. 本論文で検証する自動チューニングの手法

スーパーノードアルゴリズムは left-looking アルゴリズムのブロック化手法の一種ととらえることができる．したがって行列構造とアーキテクチャのキャッシュ

サイズによって，最適なブロック幅を探索する必要がある．

4.1 ブロック化パラメータとワーキングエリア

行列ベクトル積演算のブロック化に関わるパラメータを挙げると以下のようになる．

- スーパーノードのサイズ $t = (r - s)$.
- スーパーノードのブロック化幅 b.
- スーパーノード融合の程度 r.
- パネル幅 w.

これらのパラメータを用いると，演算カーネルである行列ベクトル積 (DGEMV(b,t)) を w 回実行するために要するデータ量 WS は式 (2) で表わせる．

$$WS = b * t + (t + b) * w + b * w \qquad (2)$$

右辺第 1 項はブロックサイズ，第 2 項は行列ベクトル積に用いるベクトルサイズ，第 3 項はデータ更新に必要なパネルサイズである．WS は一回のパネル更新でキャッシュに呼ばれるデータ量に相当する．

4.2 探索の分類

本論文では実行時に探索するパラメータと実行前に探索するパラメータを分類し，両者の最適値を組み合わせた．パラメータの割り当ては以下の通りである．

- **実行時**
  r: スーパーノード融合の程度
  w: パネル幅
- **実行前**
  b: スーパーノードのブロック幅
  t: スーパーノードのサイズ

b, t はアーキテクチャのキャッシュサイズから実行前に最適値を探索できるが，r, w は係数行列の構造によって最適値が変化するため，実行時に探索を行った．

4.3 実行時に決定するパラメータと探索法

本論文では，実行時に決定するパラメータとして実験的に次の値を選択し，最適値となる組合せを探索した．

- $r = 4, 8, 12, 16$
- $w = 4, 8, 12, 16$

4.4 実行前に決定するパラメータと探索法

WS 中の b, t は，アーキテクチャのキャッシュサイズによって決まるため．本論文では次のような手法を用いて決定した．

(1) row, column の値を変え，DGEMV(row,column) の性能を調べる．なお，スーパーノード中では行列ベクトル積は DGEMV(b,t) の形で使われる．

(2) 次に，WS がキャッシュサイズと同程度になり，かつ DGEMV(b,t) が高速に行われるような b, t を求める．

具体例は次のようになる．まずキャッシュサイズが 32KB で $w = 4$ のとき，式 (2) より b と t の関係は図 7 のようになる．図 7 から WS とキャッシュとのマッチングを図るためには，スーパーノードのサイズが大きいときにブロック幅を小さくしなければならないことがわかる．なお逆も同様である．

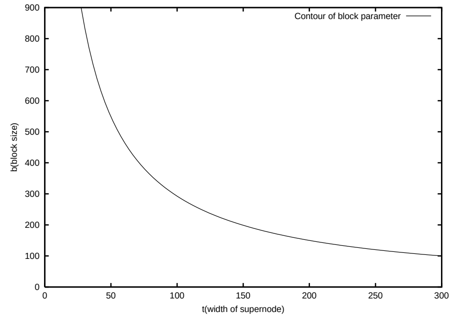

図 7：スーパーノードのサイズとブロック幅の関係．
(キャッシュサイズ=32KB，$w = 4$)

次に，DGEMV(b,t) の演算性能を b, t の値に応じて測定する．測定結果の例が図 8 である．一般にスーパーノードの幅 t は，ブロック幅 b よりも小さいので，測定範囲はスーパーノードの幅のほうが狭くなる．

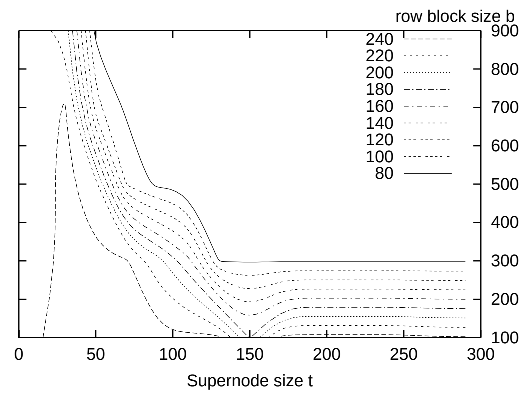

図 8：行列ベクトル積の性能 (Mflops)．
(Pentium3, compiler=gcc 2.0, compile option=-O3)

図 7 の曲線上の (b,t) の組合せの内，DGEMV(b,t) の性能が最も良い (b,t) を選択する．この例では (b,t)=(600,50) が最適値であるとわかる．

なお，(b,t) はキャッシュサイズと DGEMV の性能によってのみ決まる値であり，係数行列の構造には無関係である．したがってこの処理はプログラム実行前に一度だけ行えばよい．

5. 性能評価

5.1 実験環境

ここでは自動チューニング手法の効果を検証するため，表 1 に記した 5 つのアーキテクチャ上で性能評価を行った．表 1 中のキャッシュサイズの単位は KB である。I は命令キャッシュ，D はデータキャッシュを表す．

なお，プログラム実行前に決定するパラメータ b, t を，4.3 節の手法で探索した結果は次の通りである．w は各 b, t に対応する値である．

- **Sun Enterprise 3500**
  (w,t,b)=(4,220,136),(8,220,128),
  (12,220,120),(16,220,130)

- **COMPAQ AlphaServer GS80**
  (w,t,b)=(4,130,923),(8,270,440),
  (12,270,424),(16,270,409)
- **SGI2100**
  (w,t,b)=(4,175,345),(8,170,336),
  (12,170,319),(17,170,303)
- **HITACHI SR8000/MPP**
  (w,t,b)=(4,240,770),(8,220,806),
  (12,200,846),(16,200,813)
- **Freebsd Machine**
  (w,t,b)=(4,65,434),(8,60,414),
  (12,60,372),(16,60,337)

表 1　アーキテクチャの特性とコンパイラオプション仕様.

Name	Processor	cache size L1 (I,D)	compiler & option
Sun Enterprise 3500	Ultra Sparc2 336MHz	(16,16)	Sun Workshop Compiler 5.0 -xO3
COMPAQ—AlphaServer GS80	Alpha 21264 731MHz	(64,64)	Compaq C V6.3-027 -O3
SGI2100	MIPS R12000 350MHz	(32,32)	MIPSpro Compilers V7.30 -O3
HITACHI SR8000/mpp	RICS Micro Processor 1.8GHz	(64,128)	Optimizing C Compiler -O3
Freebsd Machine	Pentium3 E 600MHz	(16,16)	gcc2.0 -O3

5.2 実験対象

実験に用いた行列は表 2 の通りである．いずれも University of Florida Sparse Matrix Collection[13] から入手した.

表 2　テスト行列の特徴.

(No.)Name	Dimension	Nonzeros	Sparsity(%)
(1)Orani678	2529	90158	1.4
(2)Lnsp3937	3937	25407	0.16
(3)Sherman5	3312	20793	0.19
(4)Goodwin	7320	324772	0.61
(5)Dw4096	8192	41746	0.062
(6)Coater2	9540	207308	0.23
(7)Ex12	3973	80211	0.51
(8)Ex15	6867	98671	0.21
(9)Garon2	13535	390607	0.21
(10)Bayer10	13436	94926	0.053
(11)Lhr07	7337	156508	0.29

5.3 実験結果

表 3 から表 7 に実験結果を記した．なお，データの精度を向上するため実験は 5 回行い，その平均を求めた.

表中に auto-tuning と書かれているのは，基準となる設定に対する速度向上率である．パラメータの基準には [4] より $w=10, r=5, t=100, b=200$ を選択した．W/B は最低値 (Worse) と最高値 (Best) の比を表している.

表 3　Sun Enterprise 3500 での結果.

No.	time(sec) (Mflops)	(r,w,t,b)	W/B	auto-tuning
(1)	15.6(56.1)	(8,12,220,136)	1.07	1.01
(2)	0.898(45.1)	(4,8,220,128)	1.07	1.01
(3)	0.474(47.8)	default	1.03	1.00
(4)	8.62(58.7)	(4,12,220,120)	1.03	1.00
(5)	2.15(47.5)	default	1.06	1.00
(6)	19.3(52.7)	(4,16,220,130)	1.16	1.06
(7)	1.67(51.1)	(8,16,220,130)	1.04	1.00
(8)	1.15(49.6)	(4,8,220,128)	1.04	1.03
(9)	22.7(56.0)	(16,16,220,130)	1.07	1.02
(10)	0.512(23.7)	(4,4,220,136)	1.30	1.11
(11)	0.956(32.9)	(4,4,220,136)	1.10	1.03

表 4　COMPAQ AlphaServer GS80 での結果.

No.	time(sec) (Mflops)	(r,w,t,b)	W/B	auto-tuning
(1)	3.15(279.1)	default	1.11	1.00
(2)	0.222(183.6)	(4,4,130,923)	1.14	1.04
(3)	0.100(229.6)	(8,4,130,923)	1.13	1.10
(4)	1.89(267.7)	(4,12,270,424)	1.04	1.01
(5)	0.628(162.4)	(4,4,130,923)	1.08	1.04
(6)	4.53(224.3)	(4,8,270,440)	1.12	1.05
(7)	0.397(214.1)	default	1.04	1.00
(8)	0.273(209.6)	(4,4,130,923)	1.13	1.10
(9)	5.70(223.6)	(16,8,270,440)	1.07	1.00
(10)	0.147(82.4)	(4,4,130,923)	1.52	1.19
(11)	0.247(127.3)	(4,4,130,923)	1.16	1.09

表 5　SGI2100 での結果

No.	time(sec) (Mflops)	(r,w,t,b)	W/B	auto-tuning
(1)	4.93(174.7)	(8,12,170,319)	1.13	1.05
(2)	0.370(108.8)	(4,4,175,345)	1.05	1.03
(3)	0.170(137.0)	(12,4,175,345)	1.09	1.05
(4)	2.87(176.7)	(4,12,170,319)	1.01	1.00
(5)	0.782(130.3)	(4,4,175,345)	1.06	1.03
(6)	6.61(153.6)	(4,8,170,336)	1.12	1.05
(7)	0.630(135.0)	(8,4,175,345)	1.03	1.02
(8)	0.412(138.4)	(4,4,175,345)	1.07	1.06
(9)	7.43(172.0)	(16,12,170,319)	1.06	1.01
(10)	0.256(47.3)	(4,4,175,345)	1.32	1.15
(11)	0.436(72.1)	(4,4,175,345)	1.11	1.05

5.4 考　　察

行列構造, アーキテクチャによる差が大きいが, W/B で最大 1.52 倍，平均 1.15 倍程度，auto-tuning で最大 1.19 倍，平均 1.05 倍程度の性能向上が見られた. キャッシュサイズとのマッチング作業のみでこれだけの性能向上が得られることは注目に値する.

スーパーノードアルゴリズムのようなブロック化を伴う直接解法は，一般にブロック幅のパラメータに関する組み合せ数が増大するが，その設定の多くは利用者の判断に委ねられている．利用者が不適切な設定を行うと，全く性能が得られない可能性もある．例えば本論文では実験的に $r=4,8,12,16$，$w=4,8,12,16$ を選択したが，事前に $r>16, w>16$ で実験を行っ

表 6 HITACHI SR8000/MPP での結果

No.	time(sec) (Mflops)	(r, w, t, b)	W/B	auto-tuning
(1)	7.71(113.8)	(8,12,200,846)	1.11	1.00
(2)	0.530(75.9)	(4,4,240,770)	1.08	1.02
(3)	0.230(98.4)	default	1.17	1.00
(4)	4.45(113.7)	default	1.09	1.00
(5)	1.94(52.5)	(4,4,240,770)	1.21	1.18
(6)	10.1(100.6)	(4,4,240,770)	1.35	1.18
(7)	0.940(90.5)	(8,12,200,846)	1.04	1.01
(8)	0.580(98.3)	(4,8,220,806)	1.07	1.05
(9)	16.7(75.5)	(16,8,220,806)	1.14	1.00
(10)	0.310(39.1)	(4,4,240,770)	1.26	1.06
(11)	0.530(59.4)	(4,8,220,806)	1.06	1.00

表 7 Freebsd machine での結果

No.	time(sec) (Mflops)	(r, w, t, b)	W/B	auto-tuning
(1)	7.49(117.2)	(8,12,60,372)	1.26	1.01
(2)	0.450(89.5)	(4,4,65,434)	1.11	1.01
(3)	0.230(96.6)	default	1.08	1.00
(4)	4.03(125.6)	(4,16,60,337)	1.22	1.04
(5)	1.31(77.7)	(4,4,65,434)	1.16	1.11
(6)	8.67(117.2)	(4,16,60,337)	1.43	1.11
(7)	0.830(102.8)	(8,12,60,372)	1.12	1.02
(8)	0.555(102.1)	(4,4,65,434)	1.09	1.05
(9)	10.8(117.8)	(12,16,60,337)	1.32	1.04
(10)	0.240(50.0)	(4,4,65,434)	1.33	1.13
(11)	0.415(76.0)	(8,4,65,434)	1.13	1.08

たところ，0.11 倍等の著しい性能低下が見られた．実際の測定は比較的狭い範囲であったため，性能向上の上限は 1.5 倍程度に留まったが，選択範囲を広げると性能差はさらに拡大するものと思われる．

行列別に見ると，行列により性能向上の上限に差が見られる．これはアーキテクチャの種類や行列のサイズ，非零要素数に無関係であることから，行列の非零要素の位置 [13] によるものと考えられる．例えば性能向上がさほど認められない (4)，(8) の行列は，非零要素の大部分が対角成分に存在し，スーパーノードを形成しやすい構造である．このためパラメータの値によらず Mflops 値は大きい．それに対して，(1)，(10) の行列は対角成分以外に非零要素が分散するため，ブロック幅の最適化による性能向上は大きい．このことから，もし行列の構造を定量的にモデル化できれば，性能向上の割合を見積もることができ，自動チューニングに利用できる可能性がある．

r, w もアーキテクチャによらず，行列が同じなら同じ値が選択される傾向にある．同様にモデル化により，プログラム実行前に r, w の最適値を決定することが可能となる．

6. まとめと今後の予定

本論文ではブロック化を用いた LU 分解の一つであるスーパーノードアルゴリズムを紹介し，その最適化手法の一つを検討した．従来のスーパーノードアルゴリズムを始めとする直接解法の多くは，ブロック幅等の設定を利用者の判断に任せていたため，負担が大きく十分な性能が得られないケースが多かった．そこで本論文では，アーキテクチャのキャッシュサイズから最適なブロック幅を探索する手法を試みたところ，値の組合せによっては最大 1.5 倍程度の性能差が認められた．また探索は全て自動で行われるので，利用者の負担は小さい．このことからスーパーノードアルゴリズムの自動チューニングは有用な手法と考えられる．

r, w の最適値は，アーキテクチャによらず行列が同じであれば同じ値が選択される傾向にある．また性能向上の度合も行列の構造と相関があるものと考えられる．そこで行列の非零要素の位置を定量的にモデル化できれば，r, w の最適値とその性能をプログラム実行前に知ることが可能となる．そのために消去木の分枝数と各枝の大きさを解析する手法を開発中である．

本論文の対象は逐次処理のみであったが，今後は分散処理に対しても評価を行う必要がある．さらに DGEMV と DGEMM の選択を行列の疎性に応じて行う等，自動チューニングの対象も拡大していきたい．

参 考 文 献

1) Dongarra, J.J. and Duff, L.S. and Sorensen, D.C. and Van der Vorst, H.A.: “Numerical Linear Algebra for High-Performance Computers,” SIAM Press, (1998).
2) Netlib Repository at UTK and ORNL. http://www.netlib.org/.
3) HINTS Project. http://www.hints.org/.
4) Xiaoye S. Li, “Sparse Gaussian Elimination on High Performance Computers,” PhD University of California at Berkeley, 1996.
5) 小国力, 村田健郎, 三好俊郎, Dongarra, J.J., 長谷川秀彦: “行列計算ソフトウェア, ” 丸善, (1995).
6) Alan George and Esmond Ng.: “An implementation of Gaussian elimination with partial pivoting for sparse systems,” SIAM J. Sci. Stat. Comput., 6(2):390-409, 1985.
7) J.R. Gilbert and C. Moler and R. Schreiber.: “Sparse matrices in Matlab: Design and implemantation,” SIAM J. Matrix Analysis and Applications, 13:333-356, 1992.
8) C. Ashcraft and R. Grimes.: “The influence of relaxed supernode partitions on the multifrontal method,” ACM Trans. Mathematical Software, 15:291-309, 1989.
9) J. W. H. Liu.: “The role of elimination trees in sparse factorization,” SIAM J. Matrix Analysis and Applications, 11:134-172, 1990.
10) J.R. Gilbert.: “Predicting structure in sparse matrix computations,” SIAM J. Matrix Analysis and Applications, 15:62-79, 1994.
11) 寒川光: “超高速化プログラミング技法,” 共立出版, (1995).
12) I.S.Duff and J.K.Reid.:“The multifrontal solution of indefinite sparse symmetric linear equations,”ACM Transactions on Mathematical Software,9(3):302-325,(1983).
13) University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/ davis/sparse/.