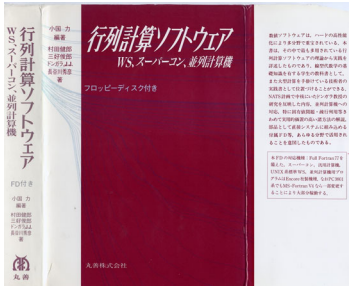
OpenMP を用いた
帯行列に対する直接解法の並列化

筑波大学　長谷川 秀彦

最近では http://www.netlib.org/utk/people/JackDongarra/la-sw.html

・連立一次方程式
　密行列　ScaLAPACK, HPL, ATLAS
　帯行列　???
　規則的疎行列　Templates
・固有値解法（対称）
　密行列　Katagiri, ScaLAPACK
　帯行列　???

行列計算ソフトウェア　1991年



話のあらすじ

- 帯行列はどこから
- 逐次コードの高速化
  （コンパイラ支援・アンローリング）
- OpenMP による並列化
- まとめ

ベクトルコンピュータ向けコード

・連立一次方程式
　密行列
　帯行列
　規則的疎行列
・固有値解法（対称）
　密行列
　帯行列

数値シミュレーションでは

連続：Partial Differential Equations;
　　Convective Diffusion Equation
$du/dt + div(-k(x,u)\nabla u + b(x,u)u) = f(x,u)$

||

離散：
　連立一次方程式 $C u' + A u = f$ を解く

必要な計算資源

	直接解法			反復解法
行列形式	密行列	帯行列	対称正定値 帯行列	非対称 7点差分 BiCG
メモリ容量	n^2	3nm	nm	7n+10n
積和演算	$n^3/3$	$2m^2n$	$m^2n/2$	30nr

帯半幅 m, 元数 n, 反復回数 r

昔話 1991年ごろ

- スーパーコンピュータ S-820/80 : 2GFLOPS
- 帯半幅 m1=150, N=22650 が限度
- 性能: m1=100, N=10100
  - 基本 567 MFLOPS (28%) … 0.71s
  - 2段 1.06 GFLOPS (53%) … 0.38s
  - 2段2行 937 MFLOPS (46%) … 0.43s
- 制約はメモリ容量だった！

必要な計算資源

	直接解法			反復解法
行列形式	密行列	帯行列	対称正定値 帯行列	非対称 7点差分 BiCG
メモリ容量	n^2 80GB	3nm 720MB	nm 240MB	7n+10n 13.6MB
積和演算	$n^3/3$ 600Tflops	$2m^2n$ 36Gflops	$m^2n/2$ 9Gflops	30np 1.8Gflops

帯半幅 m=300, 元数 n=10^5, 反復回数 r=300

その後

- S-810/20 (630M) 232 MFLOPS
- S-820/80 (2G) 1070 MFLOPS
- S-3800/480(4G) 1470 MFLOPS
- M-682H-IAP 110 MFLOPS
- M-880/310 72 MFLOPS

- Macintosh G5(8G) 1200 MFLOPS

1TFLOPS のマシンでは

	総演算量	効率	必要な時間
密行列	600Tflops	100%	600 秒
帯行列	36Gflops	1%	3.6 秒
帯行列	36Gflops	10%	0.36 秒

現在ではどれくらい？

M1=500, N=250,000, Itanium2(1.3GHz) 8CPU, 3GB

	プログラム	左辺の分解	右辺の計算 10 回
一般帯	BGLU4 & BGSLV4	41 sec (5.9G)	40 sec (190M)
右辺重視	BHLU4 & BHSLV4	43 sec (5.8G)	16 sec (461M)
対称	BSLU4 & BSSLV4	10 sec (5.8G)	10 sec (500M)

帯行列に対する直接解法

- 帯行列を対象
  （対角の両側 i-m1 ≦ j ≦ i+m1 以外はゼロ）
- 基本的にはガウスの消去法（直接解法）
- 非ゼロ要素＋Fill-in部のみをメモリに格納
  （ゼロでなくなる部分）
- 消去領域が局所的
  ー＞キャッシュにはよいが、並列性希薄

どんなマシンで？

- DELL PowerEdge 6350@筑波大:
  4 * Pentium III (550MHz; 550MFLOPS)
- HITACHI SR8000 1 node@原研:
  8 * PowerPC? (375MHz, 1.5GFLOPS)
- SGI Altix@東京大:
  32 * Itanium II (1.3GHz, 5.2GFLOPS)
- NEC SX-6/4B @原研: 4 * ?? (8GFLOPS)

帯ガウスのアルゴリズム

```
DO k = 1, N
   部分軸選択など
   DO i = k+1, min(k+m1,N)
      DO j = k+1, min(k+2*m1,N)
         a(i,j) = a(i,j)-a(i,k)*a(k,j)/a(k,k)
      END DO
   END DO
END DO
```


概要

- 帯行列はどこから
- 逐次コードの高速化
  （コンパイラ支援・アンローリング）
- OpenMP による並列化
- まとめ

帯ガウスのアルゴリズム：実際

```
DO k = 1, N
   部分軸選択など
   DO i = k+1, min(k+m1,N)
      DO j = k+1, min(k+2*m1,N)
         a(j-i,i) = a(j-i,i)-a(k-i,i)*a(j-k,k)/a(0,k)
      END DO
   END DO
END DO
```


帯ガウスのアルゴリズム：原型

```
DO k = 1, N
   部分軸選択など
   DO i = k+1, min(k+m1,N)
      DO j = k+1, min(k+2*m1,N)
         a(j-i,i) = a(j-i,i)-a(k-i,i)*a(j-k,k)/a(0,k)
      END DO
   END DO
END DO
```

アルゴリズム：スカラー T 使用

```
DO k = 1, N
    部分軸選択など
    DO i = k+1, min(k+m1,N)
       T= -a(k-i,i)/a(0,k)
       DO j = k+1, min(k+2*m1,N)
          a(j-i,i) = a(j-i,i)+T*a(j-k,k)
       END DO
    END DO
END DO
```


コード様々-1

m1=100/m1=200

原型	PE6350 (DELL/PGI)	SR8K (HITACHI/同左)	Altix (SGI/Intel)	RS6K (IBM/IBM)
配列のまま	83M 36M	252M 295M	957M 811M	284M -
スカラー T 使用	84M(15%) 35M	254M(16%) 294M	1810M(34%) 811M	348M -
1次元配列 W 使用	82M 35M	251M 288M	316M 1130M	280M -
T & W 使用	82M 35M	256M 290M	1330M 1131M	354M(22%) -

アルゴリズム：1次元配列 W 使用

```
DO k = 1, N
    部分軸選択など
    DO j = k+1, min(k+2*m1,N)
       W(j) = a(j-k,k)
    END DO
    DO i = k+1, min(k+m1,N)
       DO j = k+1, min(k+2*m1,N)
          a(j-i,i) = a(j-i,i)-a(k-i,i)*W(j)/a(0,k)
       END DO
    END DO; END DO
```


Unrolling: outmost loop k

```
DO k = 1, N, 2
   k1 = k+1
   DO i = k1+1, min(k1+m1,N)
      DO j = k1+1, min(k1+2*m1,N)
         a(j-i,i) = a(j-i,i)+T1*W1(j)+T2*W2(j)
      END DO
   END DO
END DO
```


アルゴリズム：スカラー T & 1次元配列 W 使用

```
DO k = 1, N
    部分軸選択など
    DO j = k+1, min(k+2*m1,N)
       W(j) = a(j-k,k)
    END DO
    DO i = k+1, min(k+m1,N)
       T= -a(k-i,i)/a(0,k)
       DO j = k+1, min(k+2*m1,N)
          a(j-i,i) = a(j-i,i)+T*W(j)
       END DO
    END DO
END DO
```


Unrolling: middle loop i

```
DO k = 1, N
   DO i = k+1, min(k+m1,N), 2
      DO j = k+1, min(k+2*m1,N)
         a(j-i,i) = a(j-i,i)+T*W(j)
         a(j-i-1,i+1) = a(j-i-1,i+1)+U*W(j)
      END DO
   END DO
END DO
```

Unrolling: innermost loop j

```
DO k = 1, N
  DO i = k+1, min(k+m1,N)
    DO j = k+1, min(k+2*m1,N), 2
      a(j-i,i) = a(j-i,i)+T*W(j)
      a(j+1-i,i) = a(j+1-i,i)+T*W(j+1)
    END DO
  END DO
END DO
```


コード様々-2 m1=100/m1=200

２段２行 BGLU4	PE6350 (DELL/PGI)	SR8K (HITACHI)	Altix (SGI/Intel)	RS6K (IBM)
配列のまま	110M 58M	484M 590M	553M 547M	388M -
スカラー T を使用	124M 63M	486M 595M	2030M 1920M	486M -
１次元配列 W 使用	129M 63M	518M(34%) 618M	576M 575M	420M -
T &W	145M(26%) 69M	504M 617M	2730M(52%) 2490M	553M(34%) -

Unrolling Summary * $m1^2$ N

Unrolling	Computation	LOAD	STORE	BRANCH
No Unrolling	1 / loop	4	2	2
2段: k	2 / loop	3	1	1
2行: i	2 / loop	3	2	1
2列: j	2 / loop	3	2	1
2段2行: k & i	4 / loop	2	1	0.5

逐次コードの高速化

- 並列化の前に高速化を！
- スカラー・作業配列を使うと性能アップ
  （まだまだコンパイラは頼りない！？）
- 原型のままだと　15-34% of Peak
- アンローリング：2段同時 1.1-1.9 倍
  2段2行同時 1.5-2.1 倍
- 最終的には　26-52% of Peak
- ATLAS/BLAS の LAPACK はもっとよい！

アンローリング m1=100/m1=200

	PE6350 (DELL/PGI)	SR8K (HITACHI)	Altix (SGI/Intel)	SX-6 (NEC)
原型	82M(1.0) 35M	256M(1.0) 290M	1.33G(1.0) 1.13G	1.68G(1.0) -
2段同時 k で展開	152M(1.8) 65M	492M(1.9) 593M	1.57G(1.1) 1.45G	2.60G(1.5) -
2段2行同時 k と i に展開	145M(1.7) 69M	504M(1.9) 617M	2.70G(2.0) 2.49G	2.99G(1.7) -

概要

- 帯行列はどこから
- 逐次コードの高速化
  （コンパイラ支援・アンローリング）
- OpenMP による並列化
- まとめ

OpenMP を用いた並列化

- コンパイラによる(半)自動並列化
- ループなどに並列化指示を与える
- 並列化指示は逐次コードではコメント
- 並列度は実行時の環境変数で決定
- アルゴリズムの正しさはプログラマ次第
- 安易な並列化:
  (いちばん重要な部分だけにディレクティブを)

ディレクティブはどこが効果的か

- 最内側ループへ DO Parallel
  ベクトル機では禁止的; OpenMP でも同じ?
- ひとつ外側のループへ DO Parallel
  配列参照が必須;逐次版では遅くなる
- ひとつ外側のループへ DO Parallel Private
  スカラーの使用も可能
- 最外側ループへ DO Parallel?
  不可能

OpenMP の実例

```
!$OMP Parallel Do Private(T)
      DO i = k+1, min(k+m1,N)
         T= -a(k-i,i)/a(0,k)
         DO j = k+1, min(k+2*m1,N)
           a(j-i,i) = a(j-i,i)+T*W(j)
         END DO
      END DO
```


OpenMP

PE6350 4 threads

m1	100	150	200
最内側 j	152M	209M	244M
i 配列化	299M	381M	384M
i Private	331M	434M	423M(19%)
PPGen	310M	417M	407M

特殊ケース

- コンパイラによる自動並列化:
  メーカ独自の SMP 並列化
  HITACHI, IBM, Intel
- HITACHI Parallel Program Generator による
  OpenMP ディレクティブの挿入
  (HITACHI SR8000 では同一性能?)
  この入れ方はどこでも有効か?

OpenMP

SR8K 8 threads

m1	100	150	200	400
最内側 j	0.41G	0.47G	0.71G (14%)	-
i 配列	1.10G	1.72G	2.33G	-
i Private	1.21G	1.85G	2.41G (20%)	2.6G
PPGen	1.67G	2.61G	3.58G (29%)	3.5G
自動並列化	1.93G	2.55G	3.56G	3.8G

OpenMP

SX-6/4B 4 threads

m1	100	200	300
Serial	2.99G	2.48G (31%)	3.11G
i Private	0.19G	0.17G (0.5%) 0.06 times for 4 threads	-
PPGen	0.22G	0.16G (0.5%)	-

Performance(Speedup)

100/200

Threads	1	4	8	16
PE6350	82M 145M	290M(3.5) 423M(2.9)	-	-
SR8K	185M 504M	-	1.3G(7.0)/1.5G(8.1) 2.4G(3.9)/3.5G(7.1)	-
Altix	1.1G 2.4G	1.6G(1.4) 4.2G(1.7)	1.5G(1.3) 2.9G(1.2)	1.0G(0.9) 1.5G(0.6)

OpenMP

Altix 8 threads;スカラー T 使用

m1	100	150	200	400
最内側	0.08G	0.12G	0.14G	-
i 配列	1.1G	1.6G	2.0G(6%)	-
i Private	1.2G	2.1G	2.9G(6%)	4.8G
PPGen	0.6G	1.0G	1.5G(3%)	2.9G

Performance(% of Peak)

100/200

Threads	1	4	8	16
PE6350	82M(14%) 145M(26%)	290M(13%) 423M(19%)	-	-
SR8K	185M(12%) 504M(33%)	-	1.3G(10%) 2.4G(20%)	-
Altix	1.1G(21%) 2.4G(46%)	1.1G(5%) 2.4G(11%)	1.6G(3%) 4.2G(10%)	1.5G(1%) 2.9G(3%)

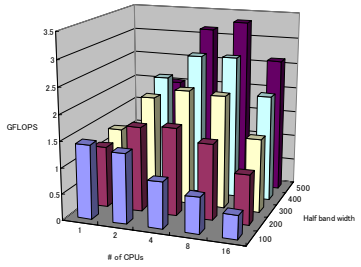
OpenMP による並列化

- 最内側は非常に遅い。使うべからず！
- Private Clause は効果的
- PPGen の性能はマシンによってまちまち
- たったひとつの directive で性能向上。cost-effective このうえなし。
- SX-6 は速すぎて並列化出番なし

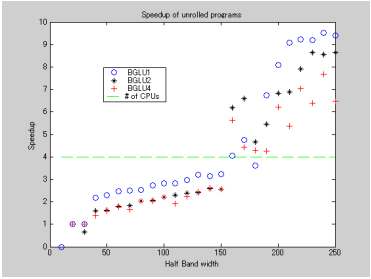
Performance(speedup) on Altix

Threads	Serial	1	2	4	8	16
m1=400	3.06G (1.15)	2.60G (1.00)	4.80G (1.84)	6.04G (2.40)	4.87G (1.84)	2.78G (1.03)
m1=500 (3.0GB)	No Memory	0.73G (1.00)	4.87G (6.57)	6.77G (9.17)	5.97G (8.08)	3.58G (4.79)

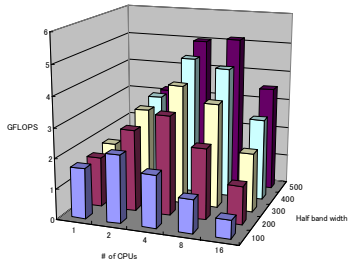
Size and Number of CPUs BGLU1



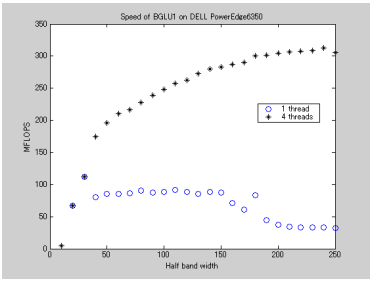
Speedups on DELL 4 CPUs



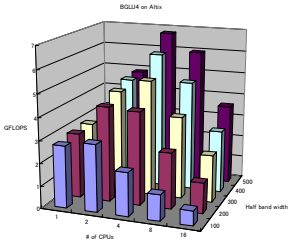
Size and Number of CPUs BGLU2



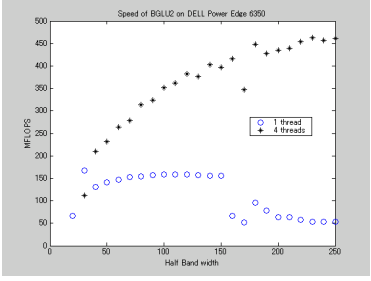
Size and Performance of BGLU1



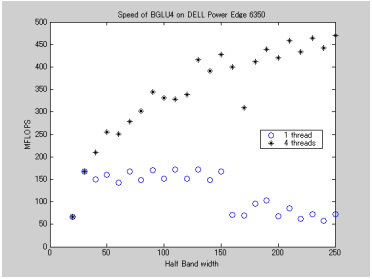
Size and Number of CPUs BGLU4



Size and Performance of BGLU2



Size and Performance of BGLU4



Information to use

Machine	Methods	m1	Decomp.	Solve #10
DELL(2.2G)	OpenMP,4	200	15s(.47G)	11s(41M)
SR8000(12G)	OpenMP,8	400	38s(2.6G)	9.7s(.39G)
SR8000	Auto, 8	400	26s(3.8G)	8.3s(.46G)
SR8000	Auto, 8	500	64s(3.8G)	13s(.55G)
SX-6(8G)	Serial, 1	300	10s(3.1G)	22s(70M)
Altix(83.2G)	OpenMP,8	500	41s(5.9G)	40s(.19G)

概要

- 帯行列はどこから
- 逐次コードの高速化
  （コンパイラ支援・アンローリング）
- OpenMP による並列化
- まとめ

これから

- この性能で許容できるか？
- もっと性能出せないの？
- （現在の）大規模問題はこれでいい？
  （性能がでないのはわかっているが、並列化は？）
- 分散並列：PC クラスタではどうか？

OpenMP's Parallelization のまとめ

- OpenMP は中間のループに使おう.
- 最適な directive マシンに依存する.
- なんてってたって簡単. cost-effective.
- Tools は完全ではないが、使える.
- 性能はどう評価するべきか?
- Not Scalable – Shared machine.
- Unrolling と Tuning は依然として必要.

インターネット時代の数学 1998 年