A Note on ScaLAPACK's Banded System Solvers

Peter Arbenz*

Abstract

We suggest modifications in the local computations of the ScaLAPACK subroutines for solving diagonally dominant and arbitrary narrow-banded linear systems. The modifications concern the way auxiliary variables are stored. The numerical properties of the algorithms are not affected. However, as the way the memory is accessed is changed the performance of the solvers is significantly improved. We discuss numerical experiments executed on the IBM SP and on a Beowulf PC cluster.

Key words: Narrow-banded systems of equations, ScaLAPACK.

AMS subject classifications: 65F05, 65Y05, 65Y20.

1 Introduction

ScaLAPACK [6] is the standard software library for solving the basic dense linear algebra problems on scalable parallel processors computers. In this note we discuss issues concerning the subroutines for solving diagonally dominant and general narrow-banded linear systems of equations.

In earlier papers [4, 5] we have compared the implementation of these algorithms in ScaLAPACK [7] with our own implementation [2, 3]. Experiments on the Intel Paragon and on the IBM SP revealed that our implementation scaled much better than the one in ScaLAPACK.

We have speculated about reasons for the different behaviors. We have claimed that, on one hand, the performance of the local computations in the ScaLAPACK implementation is degraded by an inappropriate storage scheme of the auxiliary arrays. The program was optimized for vector machines and therefore performed unsatisfactory on the RISC based machines. On the other hand, there is communication overhead implemented in ScaLAPACK that is unnecessary in situations where a single system of equations (with one or multiple right hand sides) is to be solved.

In this paper we focus on the first issue. In sections 2 and 3 we describe how we modified the storage schemes of the auxiliary arrays in the ScaLAPACK routines. The new schemes display a more favorable ratio of memory accesses vs. floating point operations and thus give rise to higher Mflop/s rates or shorter execution times, respectively. The numerical experiments conducted on an IBM SP and on a Beowulf confirm the improvements.

2 The diagonally dominant case

We first consider the case where the system matrix A, say, is diagonally dominant band matrix with lower half-bandwidth k_l and upper half-bandwidth k_u. We assume that A is narrow-banded, i.e. that $k := \max\{k_l, k_u\} \ll n$. For

*Institute of Scientific Computing, ETH Zurich, 8092 Zurich, Switzerland, arbenz@inf.ethz.ch

solving $A\mathbf{x} = \mathbf{b}$ in parallel on a p processor multicomputer we partition A according to

$$(1) \qquad A = \begin{pmatrix} A_1 & B_1^U & & & & \\ B_1^L & C_1 & D_2^U & & & \\ & D_2^L & A_2 & B_2^U & & \\ & & \ddots & \ddots & \ddots & \\ & & & B_{p-1}^L & C_{p-1} & D_p^U \\ & & & & D_p^L & A_p \end{pmatrix}, \qquad \begin{array}{l} A_i \in \mathbb{R}^{n_i \times n_i}, \quad B_i^U, D_i^L \in \mathbb{R}^{n_i \times k}, \\ C_i \in \mathbb{R}^{k \times k}, \quad B_i^L, D_i^U \in \mathbb{R}^{k \times n_i}, \\ \sum_{i=1}^{p} n_i + (p-1)k = n, \end{array}$$

as a $(2p-1)$-by-$(2p-1)$ *block-tridiagonal* matrix. The subscripts indicate the number of the processor on which the respective block is stored. The solution vector $\mathbf{x}$ and the right-hand hand side $\mathbf{b}$ are partitioned accordingly. Notice that the diagonal blocks A_i are in general much bigger than the C_i. Further notice that the rectangular blocks B_i^L, B_i^U, D_i^L, D_i^U contain only a few nonzero elements.

The part of the computation that consumes most of the time is the first step of cyclic reduction (CR) where A is (for $p = 4$) factored as

$$(2) \quad A = \begin{pmatrix} L_1 & & & & & & \\ B_1^L R_1^{-1} & I & D_2^U R_2^{-1} & & & & \\ & & L_2 & & & & \\ & & B_2^L R_2^{-1} & I & D_3^U R_3^{-1} & & \\ & & & & L_3 & & \\ & & & & B_3^L R_3^{-1} & I & D_4^U R_4^{-1} \\ & & & & & & L_4 \end{pmatrix} \begin{pmatrix} R_1 & L_1^{-1} B_1^U & & & & & \\ & T_1 & & U_2 & & & \\ & L_1^{-1} D_1^L & R_2 & L_2^{-1} B_2^U & & & \\ & V_2 & & T_2 & & U_3 & \\ & & & L_2^{-1} D_2^L & R_3 & L_3^{-1} B_3^U & \\ & & & V_3 & & T_3 & \\ & & & & & L_3^{-1} D_3^L & R_4 \end{pmatrix}$$

where

$$A_i = L_i R_i, \qquad T_i = C_i - (B_i^L R_i^{-1})(L_i^{-1} B_i^U) - (D_{i+1}^U R_{i+1}^{-1})(L_{i+1}^{-1} D_{i+1}^L),$$
$$U_i = -(D_i^U R_i^{-1})(L_i^{-1} B_i^U), \qquad V_i = -(B_i^L R_i^{-1})(L_i^{-1} D_i^L).$$

The blocks T_i, U_i, and V_i are k-by-k. They are not full if $k_l \neq k_u$. They form a block-triangular submatrix in the right factor of A in (2) with which CR is continued [3]. This (communication dominated) part of the algorithm is not the subject of this note.

The factorization $A_i = L_i R_i$, the forward substitution $L_i^{-1} D_i^L$, the backward substitution $D_i^U R_i^{-1}$ and the matrix-matrix product $(D_i^U R_i^{-1})(L_i^{-1} D_i^L)$ each cost $2k^2 n_i \approx 2k^2 n/p$ floating point operations. The remaining computations have complexity $\mathcal{O}(k^3)$.

The factors L_i and R_i are stored at the place of A_i, i.e. in a k-by-n/p array according to the LAPACK storage scheme [1, 6]. The workspace for storing the fill-ins (sometimes called 'spikes') $L_i^{-1} D_i^L$ and $D_i^U R_i^{-1}$ is arranged as n/p-by-k arrays. These matrices are computed *columnwise* in ScaLAPACK, cf. Fig. 1(a). The factor L_i or R_i is

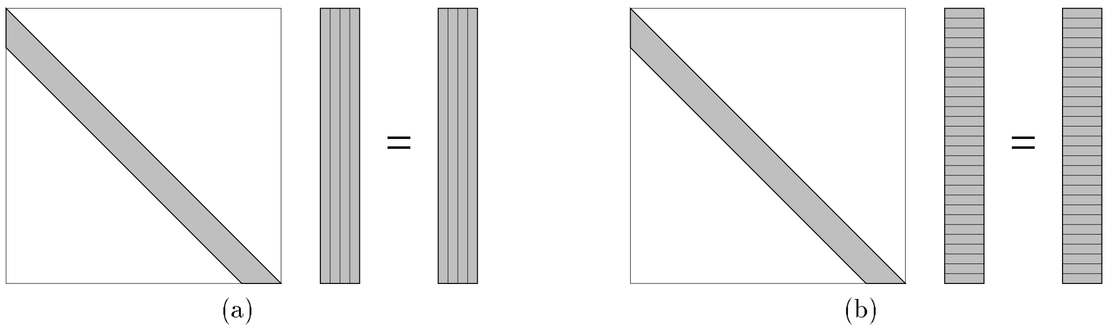

Figure 1: Storage scheme in ScaLAPACK (a) and in the modified code (b).

traversed for each right hand side. This means that for computing one element of the fill-in, a row of L_i or R_i, i.e. k floating point numbers, have to be fetched from memory and an inner product of length k has to be evaluated. (The row is multiplied with k just computed entries of the actual column of the fill-in which we assume to still be in cache.) The ratio of computation to memory accesses for this implementation is thus 2.

Our modification is simply to compute the fill-in *row-wise*, cf. Fig. 1(b), and to store it in k-by-n/p arrays. In this way a row is stored in contiguous memory locations. (The sequence of the local floating point operations is not changed.) To compute a new row of the fill-in the k just computed rows of the fill-in (a k-by-k matrix that we assume to still reside in cache) are multiplied with a row of L_i or R_i. Therefore, there are $2k^2$ flops to compute k elements of the fill-in. As k elements of L_i or R_i are read and a row of the fill-in (k elements) is stored a computation to memory access ratio of k is obtained in this way. These modifications affect only a few statements in the ScaLAPACK routines `pddbtrf.f` and `pddbtrsv.f`.

$n = 20000$	$k = 10$		$k = 20$		$k = 30$		$k = 40$		$k = 50$	
p	t_o	t_m	t_o	t_m	t_o	t_m	t_o	t_m	t_o	t_m
1	204	205	273	272	371	374	515	514	629	630
2	179	164	343	305	573	520	818	647	1126	832
4	95	88	178	160	277	246	416	331	955	423
8	53	50	136	109	226	195	350	245	488	283
12	41	37	99	80	164	142	233	188	339	245
16	32	32	77	64	124	110	152	137	259	192
24	27	27	59	50	93	81	137	110	189	146
32	23	23	48	43	76	68	110	92	150	120
48	21	22	41	37	62	56	88	76	120	100
$n = 100000$	$k = 10$		$k = 20$		$k = 30$		$k = 40$		$k = 50$	
p	t_o	t_m	t_o	t_m	t_o	t_m	t_o	t_m	t_o	t_m
1	1014	1014	1352	1352	1857	1854	2569	2568	3149	3152
2	871	803	1682	1505	2801	2439	4254	3329	5934	4360
4	444	410	855	760	1413	1234	2145	1721	4349	2762
8	232	211	456	387	1169	939	1144	1002	2397	1656
12	160	147	342	287	560	681	771	682	1542	1156
16	124	114	346	238	386	340	644	466	1155	862
24	89	83	233	188	375	328	518	404	859	617
32	70	65	176	141	280	217	451	349	637	459
48	53	51	127	107	215	168	319	253	474	354

Table 1: Execution times in milliseconds for the diagonally dominant case on the IBM SP. Best of six runs.

In Table 1 some timings are given for solving test problems with matrix orders $n = 20000$ and $n = 100000$ and half-bandwidths k ranging from 10 to 50 on an IBM SP equipped with 160 MHz P2SC processors each with 256 kB cache and 256 MB local memory. The interprocessor bandwidth is 110 MB/s. All nonzero matrix elements are 1 except the diagonal elements which were set $\alpha = 100$. The numbers show the times for solving a system of equations with a single right hand side. As problems with equal bandwidth differ only in the amount of the local computation the impact of our modifications is more prominent in the case $n = 100000$. In this case the gain through the modification improves from around 10% for $k=10$ to about 30% for $k=50$. Similar ratios t_m/t_o can be observed for the $n = 20000$. However, in this case, the gain decreases with the processor number p. This is not surprising as high processor numbers imply that the unchanged communication intensive part of the code becomes dominant. For the smallest problem sizes the impact of the modifications is relatively little as the local matrix can be hold in cache. Nevertheless, the modified code performs almost always and often significantly faster than the original code.

In Table 2 the corresponding times for a Beowulf are listed. Each node of this PC cluster consists of a dual processor 500 MHz Intel Pentium III with 512 kB level-2 cache and 1024 MB memory. The interconnection network is a switched 100 Mbit/s Ethernet. The Beowulf is twice as fast as the SP with small problem sizes. As soon as the number p of processors or the matrix bandwidth gets larger the slow memory bandwidth and the slow Ethernet connection lets the performance drop. Again, the modified ScaLAPACK routines perform better than the original routines as long as the problem size is not very small. In the latter case there is no essential difference between the two versions of the code.

3 The general case

If A is not diagonally dominant the LU factorization must be stabilized by partial pivoting. The corresponding factorization is $PA = LU$ where P is a row permutation matrix. The block-tridiagonal structure of (1) is not suited

$n=20000$	$k=10$		$k=20$		$k=30$		$k=40$		$k=50$	
p	t_o	t_m	t_o	t_m	t_o	t_m	t_o	t_m	t_o	t_m
1	109	111	208	209	340	341	500	500	1181	1183
2	187	180	617	532	1527	1458	1935	1427	3177	2513
4	109	113	411	395	608	474	980	734	1627	1280
8	92	93	176	147	326	272	511	401	836	691
12	68	66	136	116	227	200	375	299	573	507
16	61	57	114	100	183	166	294	244	432	410
24	54	54	95	84	134	135	215	198	308	324
32	54	55	85	81	117	127	186	175	252	282
$n=100000$	$k=10$		$k=20$		$k=30$		$k=40$		$k=50$	
p	t_o	t_m	t_o	t_m	t_o	t_m	t_o	t_m	t_o	t_m
1	527	531	1025	1028	1697	1700	2495	2499	5915	6034
2	848	733	2430	1662	5070	3498	9047	6272	16089	12872
4	433	392	1275	960	2801	2229	5298	4352	9788	8976
8	248	231	744	622	1775	1612	3601	3419	4198	3278
12	181	178	575	509	1434	1406	1719	1284	3275	2693
16	149	150	492	453	801	632	1322	996	2187	1753
24	117	124	288	234	561	451	929	720	1976	1693
32	106	114	228	192	434	361	726	573	1202	1005

Table 2: Execution times in milliseconds for the diagonally dominant case on the Beowulf. Best of three runs.

in this case as the narrow block rows would have to be accessed by both the two processors that hold the neighboring A_i. We therefore suggested to treat the banded matrix as a cyclic block-bidiagonal matrix [3],

$$A = \begin{pmatrix} A_1 & & & & & D_1 \\ B_1 & C_1 & & & & \\ & D_2 & A_2 & & & \\ & & \ddots & \ddots & & \\ & & & D_p & A_p & \\ & & & & B_p & C_p \end{pmatrix}, \qquad \begin{array}{ll} A_i \in \mathbb{R}^{m_i \times n_i}, & B_i \in \mathbb{R}^{k \times n_i}, \\ C_i \in \mathbb{R}^{k \times k}, & D_i \in \mathbb{R}^{m_i \times k}, \\ \sum_{i=1}^{p} m_i = n, & m_i = n_i + k, \quad k = k_l + k_u. \end{array} \tag{3}$$

The block D_1 contains the last k_l rows of A. We again discuss only the first step of the algorithm which has complexity $\mathcal{O}(k^2 n/p)$. The later steps are similar except that the matrix blocks are square [3, 4]. They have complexity $\mathcal{O}(k^3)$.

In the ScaLAPACK algorithm processor i stores the blocks A_i, B_i, C_i, D_{i+1}. The blocks B_p, C_p, and D_1 do not really appear but are incorporated into A_p. Let

$$P_i \begin{bmatrix} A_i \\ B_i \end{bmatrix} = L_i \begin{bmatrix} R_i \\ O_{k \times n_i} \end{bmatrix}, \qquad 1 \le i \le p, \tag{4}$$

be the LU factorizations of the blocks in (3), and let

$$L_i^{-1} P_i \begin{bmatrix} O_{m_i \times k} \\ C_i \end{bmatrix} = \begin{bmatrix} X_i \\ T_i \end{bmatrix}, \qquad L_i^{-1} P_i \begin{bmatrix} D_i \\ O_{k \times k} \end{bmatrix} = \begin{bmatrix} Y_i \\ V_i \end{bmatrix}, \qquad L_i^{-1} P_i \begin{bmatrix} \mathbf{b}_i \\ \boldsymbol{\beta}_i \end{bmatrix} = \begin{bmatrix} \mathbf{c}_i \\ \boldsymbol{\gamma}_i \end{bmatrix}. \tag{5}$$

Then, we factor A in the form

$$A = \begin{bmatrix} L_1^{-1}P_1 & & & \\ & L_2^{-1}P_2 & & \\ & & L_3^{-1}P_3 & \\ & & & L_4^{-1}P_4 \end{bmatrix} \begin{bmatrix} R_1 & X_1 & & & & & & Y_1 \\ O & T_1 & & & & & & V_1 \\ & Y_2 & R_2 & X_2 & & & & \\ & V_2 & O & T_2 & & & & \\ & & & Y_3 & R_3 & X_3 & & \\ & & & V_3 & O & T_3 & & \\ & & & & & Y_4 & R_4 & X_4 \\ & & & & & V_4 & O & T_4 \end{bmatrix}. \tag{6}$$

Extracting the rows in (6) that contain the k-by-k blocks T_i and V_i one obtains again a cyclic block-bidiagonal matrix

$n = 20000$	$k = 10$		$k = 20$		$k = 30$		$k = 40$		$k = 50$	
p	t_o	t_m	t_o	t_m	t_o	t_m	t_o	t_m	t_o	t_m
1	395	353	485	445	610	571	795	758	960	922
2	353	316	595	535	924	865	1176	1170	1859	1622
4	201	172	306	279	511	454	1121	637	1327	891
8	117	107	207	184	361	331	443	449	879	711
12	96	79	157	150	259	239	360	359	597	517
16	74	62	132	115	215	191	304	318	520	472
24	55	47	94	85	164	153	237	235	388	348
32	46	39	79	69	136	134	238	233	411	380
48	38	33	64	59	117	121	210	212	333	342
$n = 100000$	$k = 10$		$k = 20$		$k = 30$		$k = 40$		$k = 50$	
p	t_o	t_m	t_o	t_m	t_o	t_m	t_o	t_m	t_o	t_m
1	1978	1769	2425	2227	3050	2858	3992	3793	4801	4619
2	1749	1557	3012	2712	7052	4342	5799	5760	9456	7950
4	982	834	1478	1333	3757	2148	3019	2950	4812	4083
8	537	464	861	729	1686	1522	2276	2236	3813	2181
12	443	337	697	567	1134	1040	1539	1506	2485	2113
16	296	224	434	473	942	796	1418	1173	2534	1661
24	183	179	365	322	603	555	846	821	1382	1208
32	174	143	278	209	524	441	784	664	1356	985
48	127	102	202	183	342	312	523	496	855	732

Table 3: Execution times in milliseconds for the general case with $\alpha = 5$ on the IBM SP. Best of six runs.

$n = 20000$	$k = 10$		$k = 20$		$k = 30$		$k = 40$		$k = 50$	
p	t_o	t_m	t_o	t_m	t_o	t_m	t_o	t_m	t_o	t_m
1	353	352	443	437	567	563	755	746	920	911
2	333	296	549	511	803	839	1141	1142	1841	1599
4	207	178	340	283	683	565	1139	756	1665	1285
8	109	108	188	181	360	330	486	516	885	746
12	91	77	150	139	261	243	375	374	606	549
16	75	63	124	104	225	200	337	321	559	506
24	56	49	94	84	164	153	243	240	419	376
32	45	42	81	73	144	136	231	244	412	386
48	39	33	64	61	117	115	211	211	357	345
$n = 100000$	$k = 10$		$k = 20$		$k = 30$		$k = 40$		$k = 50$	
p	t_o	t_m	t_o	t_m	t_o	t_m	t_o	t_m	t_o	t_m
1	1765	1757	2226	2190	2845	2808	3778	3742	4631	4563
2	1644	1452	2785	2571	6909	4250	5617	5622	9263	7824
4	1010	861	1657	1504	3675	2516	3640	3564	7095	6069
8	512	451	835	772	1676	1522	1926	2224	3810	3132
12	363	341	522	583	1152	1041	1518	1512	2509	2145
16	326	258	537	464	954	790	1407	1178	2585	1666
24	231	182	349	327	612	554	857	811	1416	1155
32	176	140	281	220	520	439	763	672	1388	986
48	124	103	207	182	349	321	506	480	839	742

Table 4: Execution times in milliseconds for the general case with $\alpha = 2$ on the IBM SP. Best of six runs.

with which the factorization is pursued.

The diagonal block A_i are stored in the LAPACK storage scheme for non-diagonally dominant matrices [1, 6] in $2(k_l + k_u) + 1$ rows that store the original portions of the matrix and the fill-in that occurs in the LU factorization (4). The parallel algorithm gives rise to additional fill-in in block $\left[Y_i^T, V_i^T\right]^T$ in (5). The factorization and the computation of the additional fill-in each cost $2k^2 n/p$ flops. These are the portions of the algorithm that are rich in floating point operations. The other parts of the algorithm need fewer flops but include communication.

The modification of this algorithm was again storing the fill-in in a k-by-n/p instead of an n/p-by-k array. Only a few lines in the ScaLAPACK subroutines `pdgbtrf.f` and `pdgbtrs.f` had to be changed. We tested the impact of these changes on the IBM SP by means of the same test problems as in the previous section. By lowering the diagonal elements of the band to the values $\alpha = 5$ and $\alpha = 2$ the matrices loose the diagonal dominance. The smaller α the higher the number of permutations in the factorization phase and thus the fill-in in the upper triangular factor R_i. By consequence also the execution times increase. This can be seen from Table 3 and Table 4. The conclusions that we draw are similar to the diagonal dominant case. Here, the gain of the modifications is between 10 and 20%. It increases with the bandwidth and decreases with the number of processors for the same reasons as in the diagonally dominant case.

4 Conclusions

Changing a few lines in the Fortran codes can significantly improve the performance of ScaLAPACK's narrow-banded system solvers. As the modifications concern only the storage scheme of subroutine internal work space, the calling program is not affected by any means. Further, since the sequence of the local floating point operations is not altered the numerical properties of the modified routines are equal to those of the original routines.

We observed overall execution times for solving systems of equations with a single right hand side reduced by 10% to 30% with the solvers for diagonally dominant systems and by 10% to 20% with the solvers for general systems. Larger orders and wider bands give rise to more significant speedups.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide - Release 2.0*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1994, (Available from Netlib at URL `http://www.netlib.org/lapack/`).

[2] P. Arbenz, *On experiments with a parallel direct solver for diagonally dominant banded linear systems*, Euro-Par '96 Parallel Processing (L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, eds.), Springer, Berlin, 1996, (Lecture Notes in Computer Science, 1124), pp. 11–21.

[3] P. Arbenz and M. Hegland, *On the stable parallel solution of general narrow banded linear systems*, High Performance Algorithms for Structured Matrix Problems (P. Arbenz, M. Paprzycki, A. Sameh, and V. Sarin, eds.), Nova Science Publishers, Commack, NY, 1998, pp. 47–73.

[4] Peter Arbenz, Andrew Cleary, Jack Dongarra, and Markus Hegland, *A comparison of parallel solvers for diagonally dominant and general narrow-banded linear systems*, Tech. Report 312, ETH Zürich, Computer Science Department, January 1999, (Available at URL `http://www.inf.ethz.ch/publications/`. To appear in Parallel and Distributed Computing Practices (PCDP)).

[5] ______, *A comparison of parallel solvers for diagonally dominant and general narrow-banded linear systems II*, LAPACK Working Note 143, University of Tennessee, Knoxville, TN, May 1999, (Available at URL `http://www.netlib.org/lapack/lawns/`.

[6] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997, (Available from Netlib at URL `http://www.netlib.org/scalapack/`).

[7] A. Cleary and J. Dongarra, *Implementation in ScaLAPACK of divide-and-conquer algorithms for banded and tridiagonal systems*, Tech. Report CS-97-358, University of Tennessee, Knoxville, TN, April 1997, (Available as LAPACK Working Note #125 from URL `http://www.netlib.org/lapack/lawns/`.