A Comparison of Parallel Solvers for Diagonally Dominant and General Narrow-Banded Linear Systems II

Peter Arbenz[1], Andrew Cleary[2], Jack Dongarra[3], and Markus Hegland[4]

[1] Institute of Scientific Computing, ETH Zurich
arbenz@inf.ethz.ch
[2] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
acleary@llnl.gov
[3] Department of Computer Science, University of Tennessee, Knoxville
dongarra@cs.utk.edu
[4] Computer Sciences Laboratory, RSISE, Australian National University, Canberra
Markus.Hegland@anu.edu.au

Abstract. We continue the comparison of parallel algorithms for solving diagonally dominant and general narrow-banded linear systems of equations that we started in [2]. The solvers compared are the banded system solvers of ScaLAPACK [6] and those investigated by Arbenz and Hegland [1, 5]. We present the numerical experiments that we conducted on the IBM SP/2.

1 Introduction

In this note we continue the comparison of direct parallel solvers for narrow-banded systems of linear equations

$$A\mathbf{x} = \mathbf{b} \tag{1}$$

that we started in [2]. The n-by-n matrix A has a narrow band if its lower half-bandwidth k_l and upper half-bandwidth k_u are much smaller than the order of A, $k_l + k_u \ll n$.

We separately compare implementations of an algorithm for solving diagonally dominant and of an algorithm for solving arbitrary band systems. The algorithm for the diagonally dominant band system can be interpreted as a generalization of the well known tridiagonal *cyclic reduction* (CR), or more usefully, as Gaussian elimination applied to a symmetrically permuted system of equations $(PAP^T)P\mathbf{x} = P\mathbf{b}$. The latter interpretation has important consequences, such as it implies that the algorithm is backward stable [4]. The permutation enhances (coarse grain) parallelism. Unfortunately, it also causes Gaussian elimination to generate *fill-in* which in turn increases the computational complexity as well as the memory requirements of the algorithm [1, 6].

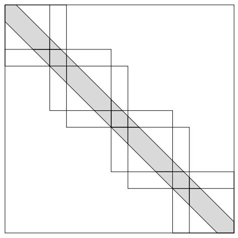
The algorithm for the arbitrary band system can be interpreted as a generalization of bidiagonal CR [10] which is equivalent to Gaussian elimination applied to a nonsymmetrically permuted system of equations $(PAQ^T)Q\mathbf{x} = P\mathbf{b}$. Here, the right permutation Q enhances parallelism, while the left permutation P enhances stability in that it incorporates the row-exchanges caused by pivoting.

Recently, the authors presented experiments with implementations of these algorithms using up to 128 processors of an Intel Paragon [2]. In this paper we complement those comparisons of the ScaLAPACK implementations with the experimental implementations by Arbenz and Hegland [1, 5] by timings on the IBM SP/2 at ETH Zurich. In section 2 we present our results for the diagonally dominant case. In section 3 the case of arbitrary band matrices is discussed. We draw our conclusions in section 4.

2 Experiments with the band solver for diagonally dominant systems

An n-by-n diagonally dominant band matrix is split according to

$$(2) \quad A = \begin{pmatrix} A_1 & B_1^U & & & & & \\ B_1^L & C_1 & D_2^U & & & & \\ & D_2^L & A_2 & B_2^U & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & B_{p-1}^L & C_{p-1} & D_p^U \\ & & & & D_p^L & A_p \end{pmatrix},$$



where $A_i \in \mathbb{R}^{n_i \times n_i}$, $C_i \in \mathbb{R}^{k\times k}$, $\mathbf{x}_i$, $\mathbf{b}_i \in \mathbb{R}^{n_i}$, $\boldsymbol{\xi}_i$, $\boldsymbol{\beta}_i \in \mathbb{R}^k$, and $\sum_{i=1}^p n_i + (p-1)k = n$, with $k := \max\{k_l, k_u\}$. The zero structure of A and its partition is depicted above to the right. This *block tridiagonal* partition is feasible only if $n_i > k$, a condition that restricts the degree of parallelism, i.e. the maximal number of processors p that can be exploited for parallel execution, where $p < (n{+}k)/(2k)$. The subscript of A_i, B_i^L, B_i^U, and C_i indicate on what processor the subblock is stored. As the orders n_i of the diagonal blocks A_i are in general much bigger than k, the order of the C_i, the first step of CR consumes most of the computational time. It is this first step of CR that has a degree of parallelism p. Therefore, if n is very big, a satisfactory speedup can be expected. After the first CR step the reduced systems are block tridiagonal with square blocks. The parallel complexity of this divide-and-conquer algorithm as implemented by Arbenz [3, 1] is

$$(3) \qquad \varphi_{n,p} \approx 2k_l(4k_u{+}1)\frac{n}{p} + \left(\frac{32}{3}k^3 + 4t_s + 4k^2 t_w\right)\lfloor \log_2(p-1)\rfloor.$$

Here, we assume that the time for the transmission of a message of n floating point numbers from one to another processor is independent of the processor distance. We represent its complexity relative to the floating point performance of the processor in the form $t_s + nt_w$ [11]. t_s denotes the startup time relative to the execution time of a floating point operation. t_w denotes the number of floating point operations that can be executed during the transmission of one word, here a 8-byte floating point number. Notice that t_s is much larger than t_w. On our target machine, the IBM SP/2 at ETH Zurich with 64 160 MHz P2SC processors, the startup time and bandwidth between applications are about 31 μs and 110 MB/s, respectively. Comparing with the 310 Mflop/s performance for the LINPACK-100 benchmark we get $t_s \approx 9600$ and $t_w \approx 22.5$. In the ScaLAPACK implementation, the factorization phase is separated from the forward substitution phase as in LAPACK. Therefore, there are more messages to be sent and the term $4t_s$ in (3) becomes $6t_s$ which can be relevant for small bandwidth. The complexity if the straightforward serial algorithm [9, §4.3] is

$$\varphi_n = (2k_u+1)k_l n + (2k_l+2k_u-1)rn + \mathcal{O}((k+r)k^2). \tag{4}$$

The computational overhead is introduced as the off-diagonal blocks D_i^L and D_i^U are filled during Gaussian elimination.

Diagonally dominant case on the IBM SP/2									
(n, k_l, k_u)	(20000, 10, 10)			(100000, 10, 10)			(100000, 50, 50)		
p	t	S	ε	t	S	ε	t	S	ε
ScaLAPACK implementation									
1	128	1.0	4e-10	618	1.0	3e+8	2558	1.0	4e+8
2	120	1.1	4e-10	580	1.1	1e+8	5660	0.45	2e+8
4	69.8	1.8	3e-10	297	2.1	1e+8	2927	0.87	2e+8
8	38.5	3.3	2e-10	155	4.0	3e-9	2010	1.2	8e-9
12	30.6	4.2	2e-10	114	5.4	2e-9	1623	1.6	7e-9
16	25.1	5.1	2e-10	86.2	7.2	2e-9	1202	2.1	6e-9
24	27.1	4.7	2e-10	69.7	8.9	2e-9	855	3.0	5e-9
32	20.7	6.2	1e-10	53.1	12	1e-9	629	4.1	4e-9
48	20.4	6.3	1e-10	40.5	15	1e-9	479	5.3	4e-9
64	12.3	10	1e-10	28.4	22	1e-9	363	7.0	3e-9
Arbenz / Hegland implementation									
1	124	1.0	4e-10	609	1.0	5e-9	2788	1.0	1e-8
2	130	0.96	4e-10	666	0.91	4e-9	4160	0.67	1e-8
4	66.6	1.9	3e-10	326	1.9	4e-9	2175	1.3	1e-8
8	37.5	3.3	2e-10	164	3.7	3e-9	1007	2.8	8e-9
12	21.8	5.7	2e-10	109	5.6	2e-9	707	3.9	8e-9
16	17.5	7.1	2e-10	83.6	7.3	2e-9	509	5.5	6e-9
24	12.0	10	2e-10	60.5	10	2e-9	374	7.5	5e-9
32	9.41	13	1e-10	47.1	13	1e-9	271	10	4e-9
48	8.25	15	1e-10	30.2	20	1e-9	199	14	4e-9
64	9.57	13	1e-10	21.7	28	1e-9	180	16	3e-9

Table 1. Selected execution times t in milliseconds, speedups $S = S(p)$, and error for the two algorithms for the three problem sizes. ε denotes the 2-norm error of the computed solution.

We compare two implementations of the above algorithm, the ScaLAPACK implementation [7] and the one by Arbenz and Hegland (AH), by means of three test-problems of sizes $(n, k_l, k_u) = (100000, 10, 10)$, $(n, k_l, k_u) = (20000, 10, 10)$, and $(n, k_l, k_u) = (100000, 50, 50)$. The matrix A always has all ones within the band and the value $\alpha = 100$ on the diagonal. The condition numbers of A vary between 1 and 3, see [2]. The right-hand sides are chosen such that the solution gets $(1, \ldots, n)^T$ which enables us to determine the error in the computed solution. We compiled a program for each problem size, adjusting the arrays to just the size needed to solve the problem on one processor.

In Tab. 1 the execution times are listed for all problem sizes. For both implementations the one-processor times are quite close. The difference in this part of the code is that the AH implementation calls the level-2 BLAS based LAPACK routine `dgbtf2` for the triangular factorization, whereas in the ScaLAPACK implementation the level-3 BLAS based routine `dgbtrf` is called. The latter is advantageous with the wider bandwidth $k = 50$, while `dgbtf2` performs (slightly) better with the narrow band.

With the bandwidth $k = 10$ problems, the ScaLAPACK implementation performs slightly faster on two than on one processor. The AH implementation slows down by 5-10%. With the large problem there is a big jump from the one- to the two-processor execution times. From (3) and (4) one sees that the parallel algorithm has a redundancy of about 4. The additional work consists of computing the fill-in and the reduced system [2] which comprises a forward elimination, a backward substitution, each with k vectors, and a multiplication of a $k \times n_i$ with a $n_i \times k$ matrix. These operations are executed at very high speed such that there is almost no loss in performance with the small and the intermediate problem. In ScaLAPACK, for forward elimination and backward substitution the level-2 BLAS `dtbtrs` is called. In the AH implementation this routine is expanded in order to avoid unnecessary checks if rows have been exchanged in the factorization phase. This avoids the evaluation of `if`-statements. In the large problem size the matrices are considerably larger. There are more cache misses, in particular in the ScaLAPACK implementation where the matrices that suffer from fill-in are stored in $n_i \times k$ arrays. In the AH implementation these matrices are stored in 'lying' arrays which increases the performance on the RISC architecture of the underlying hardware considerably [1, 8].

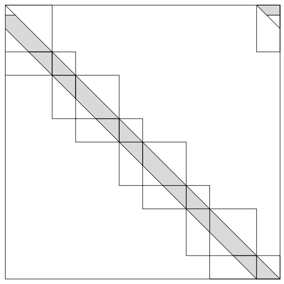
The speedups of the AH implementation *relative to the 2-processor* performance is very close to ideal for the intermediate problem. With the small problem, the communication overhead begins to dominate the computation for large processor numbers. In the large problem this is effect in not yet so pronounced. The ScaLAPACK implementation does not scale as well. For large processor numbers the difference in execution times is about 2/3 which correlates with the ratio of messages sent in the two implementations.

Clearly, the speedups for the medium size problem with large n and small k are best. The $1/p$-term that containes the factorization of the A_i and the computations of the 'spikes' $D_i^U R_i^{-1}$ and $L_i^{-1} D_i^L$ consumes five times as much time as with the small problem size and scales very well. This portion is still increased with the large problem size. However, there the solution of the reduced system gets expensive also.

3 Experiments with a pivoting solver for arbitrary band systems

The partition (2) is not suited for the parallel solution of (1) if partial pivoting is required in the Gaussian elimination to preserve stability. In order that pivoting can take place independently in block columns they must not have elements in the same row. Therefore, the separators have to be $k := k_l + k_u$ columns wide. As discussed in detail in [5, 2] we consider the matrix A as a *cyclic* band matrix by moving the last k_l rows to the top.

$$A = \left(\begin{array}{cccc|cccc} A_1 & & & & & & & D_1 \\ B_1 & & & & C_1 & & & \\ & A_2 & & & D_2 & & & \\ & B_2 & & & & C_2 & & \\ & & A_3 & & & D_3 & & \\ & & B_3 & & & & C_3 & \\ & & & A_4 & & & D_4 & \\ & & & B_4 & & & & C_4 \end{array}\right) \tag{5}$$



where $A_i \in \mathbb{R}^{m_i \times n_i}$, $C_i \in \mathbb{R}^{k \times k}$, $\mathbf{x}_i$, $\mathbf{b}_i \in \mathbb{R}^{n_i}$, $\boldsymbol{\xi}_i$, $\boldsymbol{\beta}_i \in \mathbb{R}^k$, $k := k_l + k_u$, and $\sum_{i=1}^{p} m_i = n$, $m_i = n_i + k$. If $n_i > 0$ for all i, then the degree of parallelism is p. Notice that the permutation that moves the last rows to the top is done for pedagogical reasons: it makes the diagonal blocks A_i and C_i square and the first elimination step gets formally equal with the successive ones. Also notice that A in (5) is block bidiagonal and that the diagonal blocks are lower triangular.

For solving $A\mathbf{x} = \mathbf{b}$ in parallel we apply a generalization of cyclic reduction that permits pivoting [10, 5, 2]. Its parallel complexity is

$$\varphi_{n,p}^{pp} \approx 4k^2 \frac{n}{p} + \left(\frac{23}{3}k^3 + 2t_s + 3k^2 t_w\right) \lfloor \log_2(p) \rfloor. \tag{6}$$

The serial complexity of straightforward Gaussian elimination with partial pivoting is

$$\varphi_n^{pp} \approx (2k+1) k_l n, \qquad k := k_l + k_u, \tag{7}$$

leading to a redundancy of about $2k/k_l$.

We again tested two versions of the algorithm, the ScaLAPACK implementation and the implementation by Arbenz and Hegland. We used the same test problems as above, however, we choose α, the value the diagonal elements of A, smaller. The condition number of the system matrix A, $\kappa(A)$, grows very large as α tends to one. For the problems with bandwidths 10, $\kappa(A) \approx 1$ for $\alpha = 10$ and $\kappa(A) \approx 3 \cdot 10^6$ for $\alpha = 1.01$. With the large bandwidth $k = 50$, we have

Non-diagonally dominant case on the IBM SP/2. Small problem size.												
	$\alpha = 10$			$\alpha = 5$			$\alpha = 2$			$\alpha = 1.01$		
p	t	S	ε	t	S	ε	t	S	ε	t	S	ε
ScaLAPACK implementation												
1	289	1.0	6e-10	294	1.0	3e-8	334	1.0	4e-7	333	1.0	7e-7
2	261	1.1	6e-10	274	1.1	4e-8	277	1.2	2e-6	276	1.4	7e-7
4	120	2.4	5e-10	154	1.9	4e-8	143	2.3	3e-7	136	2.4	1e-7
8	66.1	4.4	3e-10	86.3	3.4	3e-8	90.5	3.7	9e-7	90.3	3.7	2e-7
12	64.8	4.5	3e-10	62.4	4.7	4e-8	69.3	4.8	8e-7	66.1	5.0	4e-7
16	51.1	5.7	3e-10	51.5	5.7	3e-8	53.3	6.3	1e-6	52.3	6.4	7e-8
24	43.5	6.6	2e-10	41.3	7.1	9e-9	40.7	8.2	6e-7	41.0	8.1	3e-7
32	36.7	7.9	3e-10	33.9	8.7	9e-9	34.2	9.8	7e-7	34.1	9.8	1e-7
48	29.1	9.9	2e-10	29.5	10	1e-8	31.2	11	5e-7	36.8	9.1	7e-8
64	19.4	15	2e-10	19.1	15	1e-8	19.7	17	7e-7	19.9	17	5e-8
Arbenz / Hegland implementation												
1	193	1.0	7e-10	204	1.0	3e-8	242	1.0	6e-7	241	1.0	6e-7
2	171	1.1	6e-10	166	1.2	2e-8	183	1.3	1e-6	175	1.4	7e-7
4	87.6	2.2	5e-10	84.8	2.4	5e-8	95.0	2.5	1e-6	90.2	2.7	7e-7
8	48.8	3.9	4e-10	44.9	4.5	2e-8	49.4	4.9	1e-6	47.7	5.0	6e-7
12	37.1	5.2	3e-10	33.4	6.1	5e-8	35.0	6.9	1e-6	33.6	7.2	5e-8
16	30.2	6.4	3e-10	24.6	8.2	1e-8	29.7	8.1	7e-7	29.1	8.3	1e-7
24	18.7	10	3e-10	18.8	11	1e-8	21.1	11	1e-6	19.7	12	6e-7
32	15.2	13	3e-10	15.4	13	1e-8	16.3	15	5e-7	16.3	15	2e-7
48	13.8	14	3e-10	16.6	12	1e-8	13.3	18	7e-7	12.7	19	4e-8
64	11.1	17	3e-10	11.3	18	1e-8	11.7	21	3e-7	12.8	19	2e-7

Table 2. Selected execution times t in milliseconds, speedups S, and 2-norm errors ε of the two implementations for the small problem size $(n, k_l, k_u) = (20000, 10, 10)$ with varying α.

$\kappa(A) \approx 2 \cdot 10^5$ for $\alpha = 10$ and $\kappa(A) \approx 5 \cdot 10^8$ for $\alpha = 1.01$. Tables 2, 3, and 4 contain the respective numbers, execution time, speedup and 2-norm of the error, for the three problem sizes.

Relative to the AH implementation the execution times for ScaLAPACK comprise overhead proportional to the problem size, mainly zeroing elements of work arrays. This is done in the AH implementation during the building of the matrices. Therefore, the comparison in the non-diagonally dominant case should not be based primarily on execution times but on speedups. The execution times increase with the condition number $\kappa(A)$ of the problem which is of course hard or even impossible to predict as the pivoting procedure is unknown. At least the two problems with bandwidth $k = k_l + k_u = 20$ can be discussed along similar lines. (ScaLAPACK does *not* give correct results for processor numbers $p \leq 4$. This did not happen on the Intel Paragon [2]. Actually, the error occurs only on the last processor p. The execution times seem not to be affected.) The AH implementation scales better than ScaLAPACK. Its execution times

Non-diagonally dominant case on the IBM SP/2. Intermediate problem size.												
	$\alpha = 10$			$\alpha = 5$			$\alpha = 2$			$\alpha = 1.01$		
p	t	S	ε	t	S	ε	t	S	ε	t	S	ε
ScaLAPACK implementation												
1	1443	1.0	2e+9	1454	1.0	8e+11	1660	1.0	2e+12	1660	1.0	1e+10
2	1294	1.1	9e+8	1302	1.1	3e+11	1360	1.2	1e+12	1335	1.2	6e+9
4	799	1.8	9e+8	643	2.3	2e+11	685	2.4	7e+11	671	2.5	5e+9
8	412	3.5	4e-9	404	3.6	2e-6	416	4.0	1e-5	420	4.0	3e-6
12	279	5.2	4e-9	276	5.3	5e-7	290	5.7	6e-6	279	6.0	2e-6
16	210	6.9	3e-9	209	7.0	1e-6	219	7.6	6e-6	216	7.7	3e-6
24	152	9.5	3e-9	152	9.5	6e-7	151	11	5e-6	150	11	1e-6
32	115	13	2e-9	113	13	4e-7	123	14	3e-6	117	14	2e-6
48	84	17	2e-9	85.1	17	8e-7	85.9	19	2e-6	84.4	20	1e-6
64	63	23	2e-9	58.5	25	4e-7	61.8	27	2e-6	61.6	27	5e-7
Arbenz / Hegland implementation												
1	985	1.0	8e-9	978	1.0	2e-6	1252	1.0	2e-5	1173	1.0	7e-6
2	823	1.2	8e-9	826	1.2	2e-6	923	1.4	1e-5	878	1.3	5e-6
4	415	2.4	6e-9	421	2.3	5e-7	467	2.7	8e-6	439	2.7	5e-6
8	214	4.6	5e-9	213	4.6	3e-7	236	5.3	8e-6	225	5.2	2e-6
12	145	6.8	4e-9	145	6.8	9e-7	159	7.9	5e-6	152	7.7	1e-6
16	113	8.7	4e-9	109	8.9	6e-7	140	9.0	4e-6	115	10	9e-7
24	74.6	13	3e-9	75.1	13	6e-7	86.1	15	3e-6	80.3	15	1e-6
32	60.7	16	3e-9	57.3	17	1e-6	62.8	20	4e-6	62.1	19	9e-7
48	42.2	23	2e-9	41.0	24	4e-7	48.0	26	4e-6	43.3	27	6e-7
64	36.3	27	2e-9	32.4	30	6e-7	38.6	32	5e-6	34.5	34	7e-7

Table 3. Selected execution times t in milliseconds, speedups S, and 2-norm errors ε of the two implementations for the medium problem size $(n, k_l, k_u) = (100000, 10, 10)$ with varying α.

for large processor numbers is about half of that of the ScaLAPACK implementation except for $p = 64$. For a reason not yet clear to us, the ScaLAPACK implementation performs relatively fast for $p = 64$ when the execution time of the AH implementation is about 2/3 of ScaLAPACK, reflecting again the ratio of the messages sent. In contrast to the results obtained for the Paragon, the execution times for the pivoting algorithm for $\alpha = 10$ are clearly longer than for the 'simple' algorithm. Thus, the suggestion made in [4] to *always* use the pivoting algorithm can now definitively be rejected. The memory consumption of the pivoting algorithm is higher anyway. On the other hand, the overhead for pivoting in the solution of the reduced system by bidiagonal cyclic reduction is not so big that it justifies sacrificing stability.

With the large problem size, ScaLAPACK shows an extremely bad one-processor performance. In the ScaLAPACK implementation the auxiliary arrays mentioned above are accessed even in the one-processor run (when they are not needed) leading to an abundant memory consumption. The matrices do not fit

Non-diagonally dominant case on the IBM SP/2. Large problem size.												
	$\alpha = 10$			$\alpha = 5$			$\alpha = 2$			$\alpha = 1.01$		
p	t	S^*	ε	t	S^*	ε	t	S^*	ε	t	S^*	ε
ScaLAPACK implementation												
1	92674	1.0	1e+11	78777	1.0	1e+12	69759	1.0	2e+12	54908	1.0	3e+11
2	7857	0.41	4e+10	7664	0.46	4e+11	11888	0.43	6e+11	9056	0.57	1e+11
4	3839	0.85	6e+10	3924	0.90	3e+11	7102	0.73	4e+11	4630	1.1	7e+10
8	3072	1.1	1e-6	3054	1.2	9e-6	4125	1.2	2e-4	3392	1.5	7e-4
12	1993	1.6	3e-6	1991	1.8	1e-5	2754	1.9	1e-4	2272	2.3	3e-5
16	2122	1.5	2e-6	2135	1.6	2e-5	2625	2.0	1e-4	2369	2.2	3e-4
24	1174	2.8	1e-6	1123	3.1	3e-5	1476	3.5	1e-4	1252	4.1	1e-4
32	1201	2.7	9e-7	1179	3.0	3e-6	1447	3.6	8e-5	1287	4.0	4e-4
48	710	4.6	2e-6	752	4.7	3e-6	896	5.8	2e-4	763	6.8	9e-5
64	636	5.1	8e-7	732	4.8	2e-5	786	6.6	2e-4	772	6.7	9e-5
Arbenz / Hegland implementation												
1	3257	1.0	6e-6	3514	1.0	2e-5	5155	1.0	2e-4	5170	1.0	1e-3
2	6102	0.53	4e-6	5893	0.60	9e-6	10140	0.51	1e-4	7305	0.71	7e-4
4	3074	1.1	2e-6	3079	1.1	6e-5	5182	1.0	1e-4	3786	1.4	1e-4
8	1671	1.9	2e-6	1669	2.1	1e-5	2723	1.9	2e-4	2023	2.6	9e-5
12	1261	2.6	2e-6	1254	2.8	1e-5	1944	2.7	8e-5	1496	3.5	2e-4
16	1008	3.2	1e-6	1015	3.5	1e-5	1514	3.4	1e-4	1176	4.4	2e-4
24	833	3.9	3e-6	836	4.2	6e-5	1162	4.4	1e-4	945	5.5	2e-4
32	724	4.5	1e-6	710	4.9	2e-5	952	5.4	1e-4	797	6.5	2e-4
48	668	4.9	2e-6	661	5.3	2e-5	1093	4.7	1e-4	717	7.2	5e-5
64	597	5.5	1e-6	598	5.9	2e-6	724	7.1	2e-4	652	7.9	5e-5

Table 4. Selected execution times t in milliseconds, speedups S, and 2-norm errors ε of the two implementations for the large problem size $(n, k_l, k_u) = (100000, 50, 50)$ with varying α. Speedups have been taken with respect to the one-processor times of the AH-implementation.

into local memory of 256 MB any more. The ScaLAPACK run times for processor numbers larger than 1 are comparable with the AH implementation. We therefore relate them to the one-processor time of the AH implementation (a call to LAPACK's routines `dgbtrf` and `dgbtrs`) to determine speedups. The AH implementation performs quite as expected by the complexity analysis. As the band is now relatively wide, factorization and redundant computation (fill-in, formation of reduced system) perform at about the same Mflop/s rate. The fourfold work distributed over two processors results in a 'speedup' of about 0.5. In this large example the volume of the interprocessor communication is big. A message consists of a small multiple of k^2 8-byte floating point numbers (20 kB). Thus, the startup time t_s constitutes only a small fraction at the interprocessor communication cost. The latter differ only little in the ScaLAPACK and AH implementation. In this large problem size, with regard to speedups ScaLAPACK performs slightly better than the AH implementation. The execution times are however longer by about 10-20%.

4 Conclusions

The execution times measured on the IBM SP/2 are shorter than on the Intel Paragon by a factor of about five for the small and 10 for the large problems on one processor. As the Paragon's communication network has a relatively much higher bandwidth we observed better speedups on this machine which narrowed the gap [2].

For systems with very narrow band, the implementations by Arbenz and Hegland which are designed to reduce the number of messages that are communicated are faster. The difference is however not too big. The flexibility and versatility of the ScaLAPACK justifies the loss in performance. We are convinced that a few little improvements in the ScaLAPACK implementation, in particular the treatment of auxiliary arrays, will further narrow the gap.

Nevertheless, it may be useful to have in ScaLAPACK a routine that combines the factorization and solution phase as in the AH implementation. Appropriate routines would be the 'drivers' `pddbsv` for the diagonally dominant case and `pdgbsv` for the non-diagonally dominant case. In the present version of ScaLAPACK, the former routine consecutively calls `pddbtrf` and `pddbtrs`, the latter calls `pdgbtrf` and `pdgbtrs`, respectively. The storage policy could stay the same. So, the flexibility in how to apply the routines remains.

On the IBM SP/2, in contrast to the Intel Paragon, the overhead for pivoting was always noticeable. Therefore, the suggestion made in [4] to *always* use the pivoting algorithm can now definitively be rejected. The memory consumption of the pivoting algorithm is about twice as high, anyway.

References

1. P. Arbenz, *On experiments with a parallel direct solver for diagonally dominant banded linear systems*, in Euro-Par '96, L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, eds., Springer, Berlin, 1996, pp. 11–21. (Lecture Notes in Computer Science, 1124).
2. P. Arbenz, A. Cleary, J. Dongarra, and M. 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/`. Submitted to Parallel and Distributed Computing Practices (PCDP)).
3. P. Arbenz and W. Gander, *A survey of direct parallel algorithms for banded linear systems*, Tech. Report 221, ETH Zürich, Computer Science Department, October 1994. Available at URL `http://www.inf.ethz.ch/publications/`.
4. P. Arbenz and M. Hegland, *Scalable stable solvers for non-symmetric narrow-banded linear systems*, in Seventh International Parallel Computing Workshop (PCW'97), P. Mackerras, ed., Australian National University, Canberra, Australia, 1997, pp. P2–U–1 – P2–U–6.

5. ———, *On the stable parallel solution of general narrow banded linear systems*, in 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.
6. 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/`.
7. ScaLAPACK is available precompiled for the SP/2 from the archive of prebuilt ScaLAPACK libraries at `http://www.netlib.org/scalapack/`.
8. M. J. Daydé and I. S. Duff, *The use of computational kernels in full and sparse linear solvers, efficient code design on high-performance RISC processors*, in Vector and Parallel Processing – VECPAR'96, J. M. L. M. Palma and J. Dongarra, eds., Springer, Berlin, 1997, pp. 108–139. (Lecture Notes in Computer Science, 1215).
9. G. H. Golub and C. F. van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, 2nd ed., 1989.
10. M. Hegland, *Divide and conquer for the solution of banded linear systems of equations*, in Proceedings of the Fourth Euromicro Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 394–401.
11. V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing*, Benjamin/Cummings, Redwood City CA, 1994.