On Experiments with a Parallel Direct Solver for Diagonally Dominant Banded Linear Systems

Peter Arbenz

Institute of Scientific Computing, Swiss Federal Institute of Technology (ETH), CH-8092 Zürich, arbenz@inf.ethz.ch

Abstract. We report on numerical experiments that we conducted with a direct algorithm, the single width sparator algorithm, to solve diagonally dominant banded linear systems. With detailed estimations of computation and communication cost we quantitatively analyze their influence on the parallel performance of the algorithm. We report on numerical experiments executed on an Intel Paragon XP/S-22MP.

1 Introduction

In this paper we discuss an implementation of a direct method based on the single-width separator approach, also known as algebraic domain decomposition, to solve a banded diagonally dominant system of linear equations

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

The $n \times n$ matrix $\tilde{A}$ is assumed to have lower half-bandwidth r and upper half-bandwidth s, meaning

$$\tilde{a}_{ij} = 0 \qquad \text{for } i - j > r \text{ or } j - i > s. \tag{2}$$

We assume that the matrix $\tilde{A}$ has a *narrow* band, such that $r + s \ll n$. If this assumption does not hold, an algorithm for full matrices adapted to the banded matrix structure is better suited for solving the problem.

We implemented the single-width separator algorithm on the Intel Paragon, a multiprocessor computer with distributed memory architecture and powerful processing nodes supporting the MIMD programming model.

To obtain speedup numbers, we will compare our implementation of the parallel single-width separator algorithm with Gaussian elimination executed on a single processor. Gaussian elimination is the method of choice for solving (1) on serial computers [8, §4.3]. Its complexity is

$$C_{\mathrm{Gauss}}(n, r, s) \approx ((2s + 3)(r + 1) - 4)\, n \text{ flops}. \tag{3}$$

We measure complexities in flops, i.e. floating point operations. A flop is either an addition, a subtraction, a multiplication, or a division.

2 The single-width separator algorithm

The single-width separator algorithm has been investigated by many authors [4], [6], [7], [13], [14]. Johnsson [11] discussed an implementation for the Connection machine CM-2. The main difference to this paper is the modeling of the interprocessor communication. Dongarra and Johnsson [6] report on a similar implementation for the Alliant FX-8 and Sequent Balance. Modifications are discussed by Conroy [4] and Wright [14]. The latter is particularly interesting, as pivoting is introduced into the algorithm. Wright's scheme gets very cumbersome, however. If pivoting is necessary the approach by Hegland [9] is probably to be prefered. The algorithm presented here is easily modified for symmetric definite matrices.

To solve (1) on a p processor multicomputer, in the single-width separator algorithm the matrix $\tilde{A}$ and the vectors $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{b}}$ are partitioned in the form

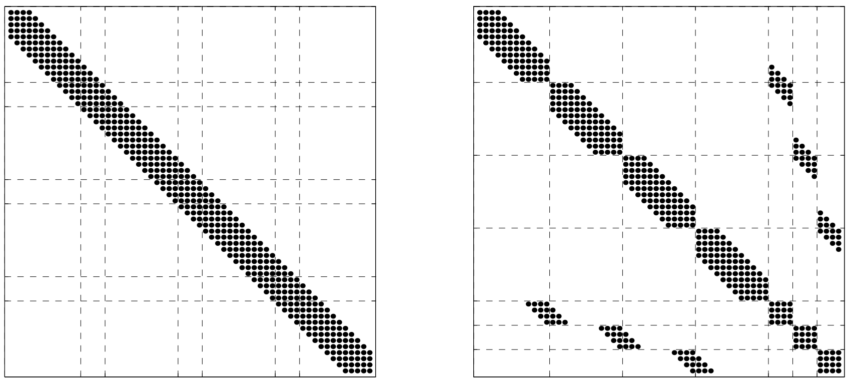
$$\begin{pmatrix} A_1 & B_1 & & & & & \\ C_1 & D_1 & C_2 & & & & \\ & B_2 & A_2 & B_3 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & C_{2p-3} & D_{p-1} & C_{2p-2} \\ & & & & B_{2p-2} & A_p \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \boldsymbol{\xi}_1 \\ \mathbf{x}_2 \\ \vdots \\ \boldsymbol{\xi}_{p-1} \\ \mathbf{x}_p \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \boldsymbol{\beta}_1 \\ \mathbf{b}_2 \\ \vdots \\ \boldsymbol{\beta}_{p-1} \\ \mathbf{b}_p \end{pmatrix}, \tag{4}$$

where $A_i \in \mathbb{R}^{n_i \times n_i}$, $B_i \in \mathbb{R}^{n_i \times k}$, $C_i \in \mathbb{R}^{k \times n_i}$, $D_i \in \mathbb{R}^{k \times k}$, $\mathbf{x}_i$, $\mathbf{b}_i \in \mathbb{R}^{n_i}$, $\boldsymbol{\xi}_i$, $\boldsymbol{\beta}_i \in \mathbb{R}^k$, $k := \max(r, s)$, and $\sum_{i=1}^{p} n_i + (p-1)k = n$. We assume that $n_i > k$ which restricts the degree of parallelism, i.e. the maximal number of processor p that can be exploited for program execution, $p < (n+k)/(2k)$. The structure of A and its submatrices is depicted in Fig. 1(a) for the case $p = 4$. The diagonal blocks A_i are band matrices with the same half-bandwidths as A itself.

Having p processors available, processor i holds matrices A_i, B_{2i-2}, B_{2i-1}, C_{2i-2}, C_{2i-1}, D_i and the vectors $\mathbf{b}_i$ and $\boldsymbol{\xi}_i$.

The single-width separator algorithm can be considered to be block-cyclic reduction [10]. In the first step, rows and columns of A in (4) are (formally) permuted in a block odd-even fashion,

$$\left[\begin{array}{ccccc|cccc} A_1 & & & & & B_1 & & & \\ & A_2 & & & & B_2 & B_3 & & \\ & & \ddots & & & & \ddots & \ddots & \\ & & & A_{p-1} & & & & \ddots & B_{2p-3} \\ & & & & A_p & & & & B_{2p-2} \\ \hline C_1 & C_2 & & & & D_1 & & & \\ & C_3 & \ddots & & & & D_2 & & \\ & & \ddots & \ddots & & & & \ddots & \\ & & & C_{2p-3} & C_{2p-2} & & & & D_{p-1} \end{array}\right] \left[\begin{array}{c} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_{p-1} \\ \mathbf{x}_p \\ \hline \boldsymbol{\xi}_1 \\ \boldsymbol{\xi}_2 \\ \vdots \\ \boldsymbol{\xi}_{p-1} \end{array}\right] = \left[\begin{array}{c} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_{p-1} \\ \mathbf{b}_p \\ \hline \boldsymbol{\beta}_1 \\ \boldsymbol{\beta}_2 \\ \vdots \\ \boldsymbol{\beta}_{p-1} \end{array}\right]. \tag{5}$$



(a) (b)

Fig. 1. Non-zero structure of (a) the original and (b) the block odd-even permuted band matrix with $n = 60$, $p = 4$, $n_i = 12$, $r = 4$, and $s = 3$.

The structure of the matrix in (5) is depicted in Fig. 1(b). We write (5) in the form

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\xi} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \boldsymbol{\beta} \end{bmatrix},$$

where the respective submatrices and subvectors are indicated by the lines in equation (5). An 'incomplete' LU factorization executed of A yields

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} L & \\ F & I \end{bmatrix} \begin{bmatrix} R & E \\ & S \end{bmatrix}, \tag{6}$$

where $A = LR$ is the ordinary LU factorization of A. The blocks in (6) are given by

$$E = L^{-1}B = \begin{pmatrix} E_1 & & & \\ E_2 & E_3 & & \\ & \ddots & \ddots & \\ & & E_{2p-4} & E_{2p-3} \\ & & & E_{2p-2} \end{pmatrix}, \quad \begin{array}{l} E_{2i-2} = L_i^{-1} B_{2i-2}, \\ E_{2i-1} = L_i^{-1} B_{2i-1}, \end{array}$$

$$F = CR^{-1} = \begin{pmatrix} F_1 & F_2 & & & \\ & F_3 & \ddots & & \\ & & \ddots & \ddots & \\ & & & F_{2p-3} & F_{2p-2} \end{pmatrix}, \quad \begin{array}{l} F_{2i-2} = C_{2i-2} R_i^{-1}, \\ F_{2i-1} = C_{2i-1} R_i^{-1}, \end{array}$$

$$S = D - FE = \begin{pmatrix} T_1 & U_1 & & & \\ V_2 & T_2 & U_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & U_{p-2} \\ & & & V_{p-1} & T_{p-1} \end{pmatrix}, \quad \begin{aligned} T_i &= D_i - F_{2i-1}E_{2i-1} - F_{2i}E_{2i}, \\ U_i &= -F_{2i}E_{2i+1}, \\ V_i &= -F_{2i-1}E_{2i-2}. \end{aligned} \tag{7}$$

The matrices E_{2i-2}, E_{2i-1}, F_{2i-2}, F_{2i-1}, V_i, T_i, U_i and the vector $\boldsymbol{\gamma}_i$ are stored in the memory of processor i. The matrices E_{2i-2}, E_{2i-1}, F_{2i-2}, and F_{2i-1} overwrite B_{2i-2}, B_{2i-1}, C_{2i-2}, and C_{2i-1}, respectively. Notice, that only E_{2i-2} and F_{2i-2} are full matrices. E_{2i-1} and F_{2i-1} keep the structure of B_{2i-1} and C_{2i-1}, respectively.

Using the factorization (6), we obtain

$$\begin{bmatrix} R & E \\ & S \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\xi} \end{bmatrix} = \begin{bmatrix} L & \\ F & I \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{b} \\ \boldsymbol{\beta} \end{bmatrix} = \begin{bmatrix} L^{-1} & \\ -FL^{-1} & I \end{bmatrix} \begin{bmatrix} \mathbf{b} \\ \boldsymbol{\beta} \end{bmatrix} =: \begin{bmatrix} \mathbf{c} \\ \boldsymbol{\gamma} \end{bmatrix}, \tag{8}$$

where the sections $\mathbf{c}_i$ and $\boldsymbol{\beta}_i$ of the vectors $\mathbf{c}$ and $\boldsymbol{\beta}$ are given by

$$\mathbf{c}_i = L_i^{-1}\mathbf{b}_i, \qquad \boldsymbol{\gamma}_i = \boldsymbol{\beta}_i - F_{2i-1}\mathbf{c}_i - F_{2i}\mathbf{c}_{i+1}.$$

Each processor can work independently on its block row computing E_{2i-2}, E_{2i-1}, F_{2i-2}, F_{2i-1}, and $\mathbf{c}_i$. Furthermore, each processor computes its portion of the matrix and right hand side of the *reduced system* $S\boldsymbol{\xi} = \boldsymbol{\gamma}$,

$$\begin{bmatrix} -F_{2i-2}E_{2i-2} & -F_{2i-2}E_{2i-1} \\ -F_{2i-1}E_{2i-2} & D_i - F_{2i-1}E_{2i-1} \end{bmatrix} \in \mathbb{R}^{2k \times 2k} \quad \text{and} \quad \begin{bmatrix} -F_{2i-2}\mathbf{c}_i \\ \boldsymbol{\beta}_i - F_{2i-1}\mathbf{c}_i \end{bmatrix} \in \mathbb{R}^{2k},$$

respectively. Until this point of the algorithm, there is no interprocessor communication.

The matrix S in the reduced system is a block tridiagonal matrix of order $(p-1)k$ with $k \times k$ blocks. The blocks are not full if $r < k$ or $s < k$. The reduced system is diagonally dominant and could be solved by block Gaussian elimination. However, for good performance on multicomputers the system $S\boldsymbol{\xi} = \boldsymbol{\gamma}$ must be solved by *block cyclic reduction* [2], i.e., the reduction step described above in equations (4) to (8) is repeated until a $k \times k$ system of equations remains. As the order of the actual system is halved in each reduction step, $\lfloor \log_2(p-1) \rfloor$ of them are needed until a full $k \times k$ system is left which is solved by ordinary Gaussian elimination.

As soon as the vectors $\boldsymbol{\xi}_i$, $1 \le i < p$, are known, each processor can compute its section of $\mathbf{x}$,

$$\begin{aligned} \mathbf{x}_1 &= R_1^{-1}(\mathbf{c}_1 - E_1\boldsymbol{\xi}_1), \\ \mathbf{x}_i &= R_i^{-1}(\mathbf{c}_i - E_{2i-2}\boldsymbol{\xi}_{i-1} - E_{2i-1}\boldsymbol{\xi}_i), \qquad 1 < i < p, \\ \mathbf{x}_p &= R_p^{-1}(\mathbf{c}_p - E_{2p-2}\boldsymbol{\xi}_{p-1}). \end{aligned}$$

In this *back substitution phase* each processor can proceed independently without interprocessor communication.

Assuming for simplicity that $k := r = s \ll n$, the *parallel* complexity of the single width separator algorithm is [2]

$$C_{\rm sws}^{\rm par} \approx 8k^2\frac{n}{p} + \left(\frac{26}{3}k^3 + 4\sigma + 4k^2\tau\right)\lfloor \log_2(p-1)\rfloor + \frac{2}{3}k^3 - \varphi(p-1)4k^3 \text{ flops}, \quad (9)$$

where

$$\varphi(p) = \begin{cases} 1, & 2^{\lfloor \log_2(p)\rfloor} \le p < \frac{3}{2}\cdot 2^{\lfloor \log_2(p)\rfloor}, \\ 0, & \frac{3}{2}\cdot 2^{\lfloor \log_2(p)\rfloor} \le p < 2\cdot 2^{\lfloor \log_2(p)\rfloor}. \end{cases}$$

If $\varphi(p) = 1$, the root node in the cyclic reduction receives (and processes) data only from one node. The influence of $\varphi(p)$ is clearly visible in the timings, cf. Fig. 4. In (9), we assumed that the time for the transmission of a message of length n floating point numbers from one to another processor can be represented in the form

$$\sigma + n\tau.$$

σ denotes the startup time *relative* to the time of a floating point operation, i.e. the number of flops that can be executed during the startup time. τ denotes the number of floating point operations that can be executed during the transmission of one (8-Byte) floating point number. On the Paragon the transmission of m bytes takes about $0.11 + 5.9\cdot 10^{-5}m$ msec. The bandwidth between applications is thus about 68 MB/s. Comparing with the 10 Mflop/s performance for the LINPACK benchmark [5] we get $\sigma = 1100$ and $\tau = 4.7$ for the Paragon. Dividing (3) by (9) and dropping lower order terms, the speedup becomes

$$S_{\rm sws}(n,k,p) = \frac{C_{\rm Gauss}(n,k)}{C_{\rm sws}^{\rm par}(n,k,p)} \approx \frac{p}{4 + \left(\frac{13k}{3} + 2\tau + \frac{2\sigma}{k^2}\right)\frac{p\lfloor \log_2(p-1)\rfloor}{n} + \frac{pk(1-6\varphi(p-1))}{3n}},$$

The processor number for which highest speedup is observed is $\mathcal{O}(n/k)$ [2]. Speedup and efficiency are relatively small, however, due to the high redundancy of the parallel algorithm.

3 Experiments

3.1 Single node performance

Most of the work on a single processor i, say, goes into the factorization of A_i, $A_i = L_i R_i$, the forward substitutions $E_{2i-2} = L_i^{-1}B_{2i-2}$ and $F_{2i-2} = C_{2i-2}R_i^{-1}$, and in the local portion $F_{2i}E_{2i}$ of T_i. Each of these computations costs approximately $2k^2 n_i$ flops, $n_i \approx n/p$.

To get performance estimates, we measured the times for solving the linear system $AX = Y$ with k right-hand sides stored in Y. Here, A is a banded, diagonally dominant matrix with equal lower and upper half-bandwidth $k = r = s$. We timed each of the three steps of the algorithm: factorization $A = LU$ of A, simultaneous forward and backward substitution. The Fortran codes were optimized for the 150-processor Intel Paragon XP/S-22MP at ETH Zurich.

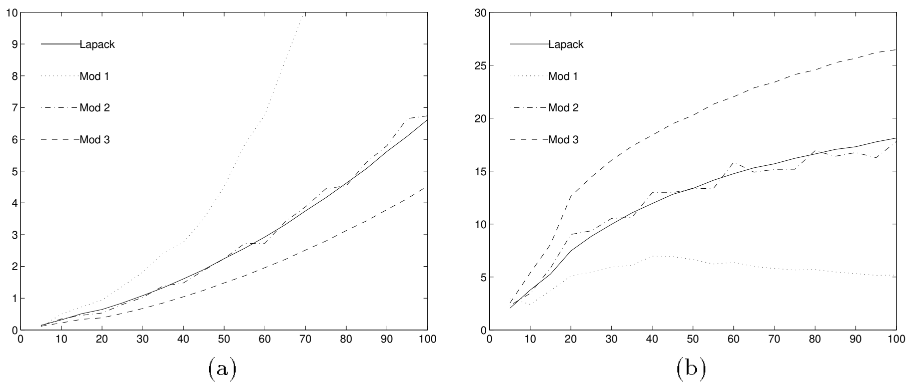


Fig. 2. Times in seconds (a) and performance in Mflop/s (b) for solving a banded system of order 2000 with varying band width/number of right sides k, $5 \leq k \leq 100$.

In a first approach, we used LAPACK [1] subroutines (`dgbtf2, dtbsv`) in a straightforward way. Unfortunately, there is no special routine for factoring banded diagonally dominant matrices in LAPACK. All non-symmetric matrices are factored by the same routine. For diagonally dominant matrices this leads to overhead due to pivot searching and unneeded memory space. Furthermore, there are no routines for forward and backward substitution with multiple right hand sides. A loop over the right hand sides is performed instead. The whole matrix is fetched from memory over and over again causing unnecessary memory traffic and possibly degradation of performance.

In a first modification, we replaced the LAPACK routines by hand-written Fortran programs. The new subroutine handled multiple right hand sides. Factorization and forward substitution were performed in one sweep. This tight integration was found to be advantageous only for very small half-bandwidths in which case the complete 'active part' of the matrix A_i could be hold in cache. For larger half-bandwidths it was better to separate the factorization from the computation of E_{2i-2} and F_{2i-2}.

In a second modification, the $n \times k$ matrices B_{2i} and E_{2i} were stored in transposed form. As these matrices are accessed row-wise, the transposition sped up the simultaneous forward and backward substitution as contiguous memory locations are accessed [3]. The transposition was found to be beneficial for the computation $T_i = F_{2i}E_{2i}$ as well. For similar reasons, A was stored in transposed form, although performance improved only little.

In a third modification, doubly nested do-loops in the factorization and the forward/backward steps were replaced by calls to BLAS-2 routines (`dger`, `dgemv`).

Figure 2(a) shows plots of measurements of the performance of the four approaches for solving $AX = Y$. The order of A was held fixed at $n = 2000$. k, the half-bandwidth of A and at the same time number of right sides, varies from 5 to 100 in steps of 5. It was found that the times behave almost linear in n as long as n is not too small. The plots show that the compiler can produce very effective code if the data is distributed properly. To get highest performance, calls to the BLAS seem to be indispensable. We attribute the good performance of the LAPACK implementation to the high-performing BLAS they are calling.

The Mflop/s rates in Fig. 2(b) are obtained by assuming a flop count of $6k^2n$ for solving $AX = Y$. The nominal peak performance of an Intel Paragon processor is 50 Mflop/s. (An MP node has two compute processors but we only used one of them.) To get more insight how the three steps, factorization, forward and backward substitution, behave we timed them independently with our fastest implementation, cf. Fig. 3. Compared with other machines the curves for

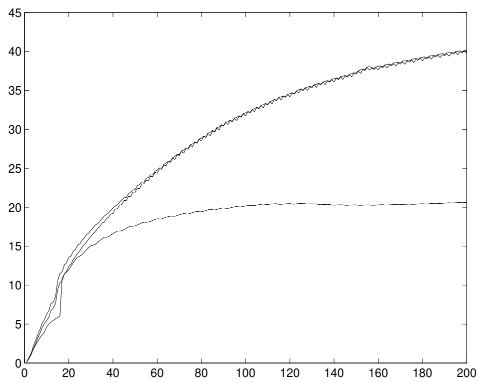


Fig. 3. Mflop/s rates for solving banded systems on the Intel Paragon. Highest performance is obtained with forward substitution, lowest performance for the factorization. The numbers are obtained with parameters $n = 1500$ and k ranging from 1 to 200.

the Paragon are quite smooth. The Mflop/s rates for forward and backward substitution tend to the processor's peak performance, however very slowly. In the factorization only about half of this performance is observed. The reason is the higher number of memory accesses caused by the rank-1 updates (LAPACK's `dger`) which were used in the LU factorization. Each call of `dger` causes the update of k^2 numbers. Forward and backward substitution were coded with calls to the LAPACK matrix-vector multiply `dgemv`, which has the same operation count as `dger` but stores only one k-vector per invocation. (If forward or backward solve are implemented with calls to `dger` the performance is reduced to the one of factorization.)

We modeled the Mflop/s rate $r(k)$ by

$$r(k) = \frac{k}{c_1 + c_2 k^{\frac{1}{2}} + c_3 k}. \tag{10}$$

The constants c_i are obtained from a least squares fit, $\tilde{r}(k)(c_1 + c_2 k^{1/2} + c_3 k) \approx k$, where $\tilde{r}(k) = 2nk^2/\tilde{t}(k)$ is obtained directly from the time measurements $\tilde{t}(k)$, cf. Tab. 1. The execution time is then estimated by

$$t(n,k) = 2nk^2/r(k) = 2nk(c_1 + c_2 k^{\frac{1}{2}} + c_3 k) \text{ msec.} \tag{11}$$

The $k^{\frac{1}{2}}$-term in (10) is not present in the well-known performance models [10, §1.3]. We found the term useful in situations where $r(k)$ was not monotonically increasing. It is possible to derive such numbers as r_∞ or $k_{1/2}$. Figure 3 indicates that the Mflop/s rates behave differently for $k \lesssim 16$ and $k \gtrsim 16$. This is probably due to the implementation of the BLAS routines. The numbers in Tab. 1 show

	c_1 [msec/flop]	c_2 [msec/flop]	c_3 [msec/flop]	$r(200)$ [Mflop/s]	$k_{1/2}$
factorization $k \leq 16$	$4.83 \cdot 10^{-3}$	$-1.94 \cdot 10^{-3}$	$3.50 \cdot 10^{-4}$		
factorization $k > 16$	$1.56 \cdot 10^{-3}$	$-2.31 \cdot 10^{-4}$	$5.74 \cdot 10^{-5}$	20.6	17
forward solve $k \leq 14$	$4.29 \cdot 10^{-3}$	$-1.74 \cdot 10^{-3}$	$2.78 \cdot 10^{-4}$		
forward solve $k > 14$	$1.18 \cdot 10^{-3}$	$1.84 \cdot 10^{-5}$	$1.75 \cdot 10^{-5}$	40.2	41
backward solve $k \leq 14$	$4.96 \cdot 10^{-3}$	$-2.01 \cdot 10^{-3}$	$3.15 \cdot 10^{-4}$		
backward solve $k > 14$	$1.48 \cdot 10^{-3}$	$-3.43 \cdot 10^{-5}$	$1.99 \cdot 10^{-5}$	39.8	43
backward solve (1 rhs)	$4.05 \cdot 10^{-4}$	$1.65 \cdot 10^{-4}$	$1.50 \cdot 10^{-4}$	6.1	6
`dgemm` $k \leq 24$	$5.86 \cdot 10^{-4}$	$-1.45 \cdot 10^{-4}$	$3.45 \cdot 10^{-5}$		
`dgemm` $k > 24$	$9.25 \cdot 10^{-5}$	$-5.55 \cdot 10^{-6}$	$2.20 \cdot 10^{-5}$	45.2	12

Table 1. Constants c_i in (10) for an i860 node of the Intel Paragon. $k_{1/2}$ is the smallest integer such that $r(k_{1/2}) > r(200)/2$.

that the performance for forward and backward substitution and in particular for matrix multiplication is close to the nominal peak performance of the node. On RISC workstations, the performance drops by a factor 2 or 3 if the size of the 'active part' of the data exceeds the cache size. Although the Paragon has a cache, in optimized code, parts of the data can 'stream' around it directly into the registers. (Cache streaming is not possible in the Paragon multiprocessor mode.)

The multiplication $F_{2i}E_{2i}$ is performed in the BLAS subroutine `dgemm`. Both matrices E_{2i-2} and F_{2i-2} are stored in a $k \times n$ array, the former in transposed form. As n is quite large in our examples, $n \geq 100$, the performance of `dgemm` depends only on k. In Tab. 1 we also give constants for backward substitution with one instead of k right sides. With them it is easy to verify that the ratio of the execotion times for the two tasks is much smaller than k [12].

3.2 Cyclic reduction

The difficult part of the algorithm is the block-cyclic reduction of the reduced system $S\boldsymbol{\xi} = \boldsymbol{\gamma}$, cf. (7). This is the only part of the algorithm with interprocessor communication. We measured the plain cyclic reduction on a varying number of processors p. The size of the blocks of the reduced system was k. The size pk of the problem increased with the number of processors. The matrices T_i, V_i and U_i together with the right side $\boldsymbol{\gamma}_i$ are stored in one array of dimension $k \times 3k{+}1$ in the form $[T_i, V_i, \boldsymbol{\gamma}_i, U_i]$. With this arrangement, data that has to be sent always resides in contiguous memory locations. Therefore, messages do not have to be collected in a buffer before being sent. In Fig. 4 timings of measurements with

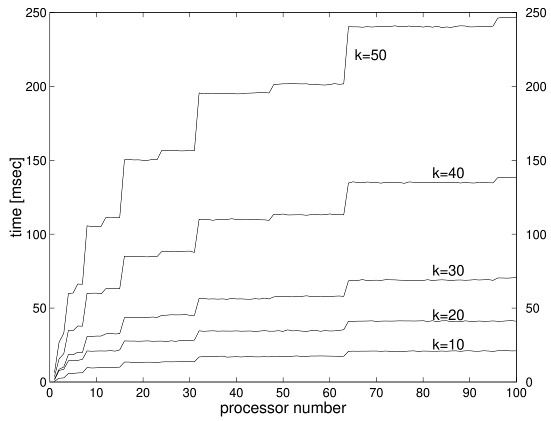


Fig. 4. Times in msec vs. processor number p for cyclic reduction of a system of order pk.

$k = 10, 20, 30, 40$, and 50 are shown. They have been obtained on the Paragon using the MPI message passing library. We model the execution time by

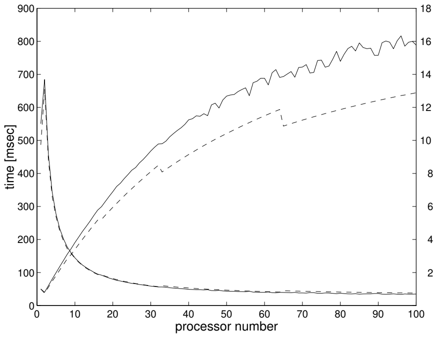
$$t(p, k) = (c_4 + c_5\ k^3)\lfloor \log_2(p) \rfloor + c_6 k^3 \varphi(p) + c_7 k^3. \tag{12}$$

A least squares fit with the data depicted in Fig. 4 gave the constants [msec]

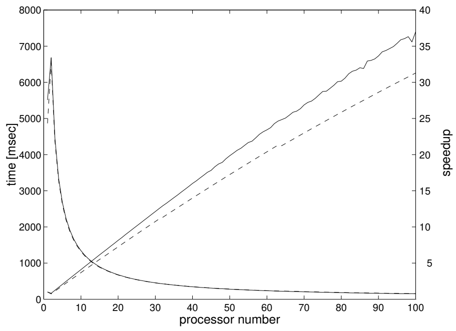
$$c_4 = 3.82, \quad c_5 = 3.17 \cdot 10^{-4}, \quad c_6 = -3.90 \cdot 10^{-5}, \quad c_7 = -1.28 \cdot 10^{-4}. \tag{13}$$

3.3 The overall problem

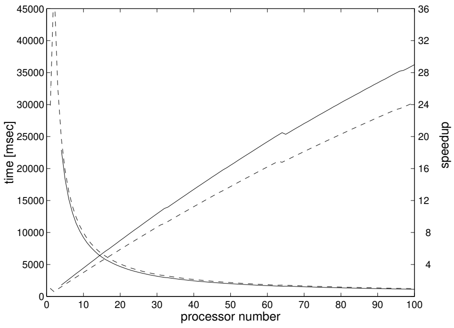
To get a rough estimate for the solution time of the overall system we sum the times obtained from equations (11) and (12) with the constants given in Tab. 1 and in (13). For $p{=}1$ only factorization and the backward substitution with one right side are taken into account. Notice, that the time for forward substitution of $\mathbf{b}$ has been neglected. Forward substitution takes place during factorization such that the matrix A_i is traversed onle once. Furthermore, the times do not comprise the summation of parts of the diagonal blocks T_i of S in (7) which have been formed by different processors.



(a)



(b)



(c)

Fig. 5. Measured (—) and estimated (- - -) execution times and speedups for the Intel Paragon for the problem sizes (a) $(n, k) = (10000, 10)$, (b) $(n, k) = (100000, 10)$, and (c) $(n, k) = (100000, 50)$. p is ranging from 1 to 100.

In Fig. 5 the actual timings $t(n, k, p)$ on the Intel Paragon XP/S-22MP for three different problem sizes are compared with the corresponding estimated times $t_{\text{est}}(n, k, p)$. Also the estimated speedups

$$s_{\text{est}}(n, k, p) := t_{\text{est}}(n, k, 1)/t_{\text{est}}(n, k, p)$$

are included. As the largest problem was too large to be solved on a single processor we approximated $t(100000, 50, 1) \approx 4 \cdot t(25000, 50, 1)$.

The estimated times are quite good, in particular those for the higher processor numbers. However, the one-processor time was in all cases underestimated by 10 to 20%. This seems to be the principal reason for the too low speedup estimations. Neglecting the forward substitution accounted for only a few percents of the error. Nevertheless, time and speedup estimation reflect the true behavior of the algorithm very well. The only exception is the too prominent appearance of the $\log(p)$ term of the cyclic reduction for the small problem size $(n, k) = (10000, 10)$.

4 Conclusions

We have shown that the execution time of the single width separator algorithm for solving banded systems of linear equations can be estimated reasonably well by a careful analysis of the important components of the algorithm. This makes it possible to predict the speedup that is to be expected if a certain number of processors is employed to solve a problem of a certain size. As the speedup cannot increase unboundedly for a fixed problem size, these estimates make it possible to determine the processor number with which the problem is solved fastest or with a desired speedup or turnaround time.

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*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1992.
2. P. Arbenz and W. Gander. A survey of direct parallel algorithms for banded linear systems. Tech. Report 221, ETH Zürich, Computer Science Department, November 1994. `ftp://ftp.inf.ethz.ch/doc/tech-reports/1994/221.ps`.
3. David H. Bailey. RISC microprocessors and scientific computing. In *Supercomputing '93*, pages 645–654, Los Alamitos, CA, 1993. IEEE Computer Society Press.
4. J. M. Conroy. Parallel algorithms for the solution of narrow banded systems. *Appl. Numer. Math.*, 5:409–421, 1989.
5. J. J. Dongarra. Performance of various computers using standard linear equation software equations software. Tech. Report CS-89-85, University of Tennessee, Computer Science Department, Knoxville, TN, November 1995.
6. J. J. Dongarra and L. Johnsson. Solving banded systems on a parallel processor. *Parallel Computing*, 5:219–246, 1987.
7. J. J. Dongarra and A. H. Sameh. On some parallel banded system solvers. *Parallel Computing*, 1:223–235, 1984.
8. G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 2nd edition, 1989.
9. M. Hegland. Divide and conquer for the solution of banded linear systems. Research report, Computer Sci. Lab. and Centre Math. Appl, Australian National University, Canberra, Australia, 1995. To be published in the Proceedings of the 4th EuroMicro Workshop on Parallel and Distributed Processing, Braga, Portugal, 24-26 January, 1996.
10. R. W. Hockney and C. R. Jesshope. *Parallel Computers 2*. Adam Hilger, Bristol, 1988.
11. S. L. Johnsson. Solving narrow banded systems on ensemble architectures. *ACM Trans. Math. Softw.*, 11:271–288, 1985.
12. A. Sameh. Parallel algorithms for structured sparse linear systems. Talk at the ICIAM'95, Hamburg, 3.–7. July 1995.
13. A. Sameh and D. Kuck. On stable parallel linear system solvers. *J. ACM*, 25:81–91, 1978.
14. S. J. Wright. Parallel algorithms for banded linear systems. *SIAM J. Sci. Stat. Comput.*, 12:824–842, 1991.

This article was processed using the LaTeX macro package with LLNCS style