Parallel Reduction to Hessenberg Form – a ScaLAPACK-based Implementation in CONLAB

Mikael Rännar
mr@cs.umu.se

Department of Computing Science
Umeå University
SE-901 87 Umeå
Sweden

Report UMINF-99.07
ISSN-0348-0542

Abstract

CONLAB offers an interactive desktop environment for parallel algorithm design and performance predictions. CONLAB is a MATLAB-style environment for developing parallel algorithms and for simulating their performance on different parallel architectures. Each parallel architecture model is defined through a few parameters, including the cost for one floating point operation, the start-up penalty for messages and a per-item cost in the message passing communication. The design of a non-trivial parallel algorithm in CONLAB is described, using the Hessenberg reduction algorithm from ScaLAPACK as illustration. We also show how the performance of the ScaLAPACK algorithm running on an IBM SP system can be simulated using the facilities offered by CONLAB. A toolbox of routines that are used as building blocks in ScaLAPACK are implemented in CONLAB including several BLACS communication routines and PBLAS computational kernels. Since the simulations performed in CONLAB runs on a Unix workstation there are limitations on the problem sizes and the processor grid sizes to be used. The main problem is the lack of memory and in order to be able to solve realistic problems in CONLAB we propose and illustrate a down-sizing and scaling approach for emulation of the execution of large scale problems. The problem is down-sized to fit the workstation environment. The parameters of the CONLAB timing model, which is a user-defined function, is scaled in concert so that we simulate the real problem as a down-sized problem in CONLAB.

Contents

1 Introduction

CONLAB (CONcurrent LABoratory) [12, 14, 15] is a MATLAB-style environment for developing algorithms for parallel architectures and for simulating different parallel architectures. The application area is matrix computations which is basic in scientific and engineering computing.

ScaLAPACK (Scalable Linear Algebra PACKage) [8, 4, 3] is a software library for linear algebra computations on distributed-memory MIMD computers. Today, ScaLAPACK includes a subset of the routines in LAPACK [1]. The programming model of ScaLAPACK is Single-Program-Multiple-Data (SPMD) using explicit message passing for interprocessor communication. The logical architecture model is a 2D mesh and matrices are assumed to be distributed over the mesh using a 2D block cyclic decomposition. The basic building block of the ScaLAPACK library is the PBLAS [7, 5], a distributed-memory version of the level 1, 2, and 3 Basic Linear Algebra Subprograms (BLAS). The PBLAS are in turn built on the BLAS for computation on single nodes and on a set of Basic Linear Algebra Communication Subprograms (BLACS) [10] for communication. The goals of the ScaLAPACK project are efficiency, scalability, reliability, portability, and flexibility.

In an earlier paper [16] we presented an algorithm development methodology for distributed memory multicomputers (DMM), using matrix multiplication and QR factorization as example algorithms developed in CONLAB. In this paper we investigate the possibility to develop, implement and evaluate a more challenging algorithm in CONLAB. We have chosen the reduction of a real general matrix, A, to upper Hessenberg form H by an orthogonal similarity transformation, $Q^TAQ = H$. A matrix H is in upper Hessenberg form if $h_{ij} = 0$ whenever $i > j+1$. This reduction is typically a first step in the solution of the standard eigenvalue problem. But it is also important in its own right, for example, when solving matrix systems of the type $(A - sI)x = b$ for many different values of the scalar s and possibly different right hand sides. The Hessenberg reduction is more complicated to parallelize than matrix factorizations like QR or LU because here there are transformations to be applied from both the left and the right hand sides of A.

As our starting point we have used the ScaLAPACK routine for the Hessenberg reduction. First we did some performance tests of this routine on an IBM SP, and then we developed an implementation of the same algorithm in CONLAB with the goal to simulate the same architecture-algorithm combination in CONLAB. The normal way to go would be the other way around, i.e., to start to develop and prototype an algorithm in CONLAB and then go on to make an efficient implementation in say Fortran or C. The long-term goal is to make this translation automatically (see [11]). Another goal with this work is to make the building blocks of ScaLAPACK available in CONLAB for future algorithm development.

The rest of the paper is outlined as follows. In Section 2 we give an introduction to the CONLAB environment and in Section 3 the sequential as well as the parallel Hessenberg reduction are described together with some performance results from the parallel implementation in ScaLAPACK. Section 4 discusses the implementation of the Hessenberg reduction in CONLAB, describing problems and solutions and also some results from test runs. Finally, in Section 5 some conclusions about the work are given.

2 CONLAB Environment

CONLAB is an environment for developing algorithms for parallel computer architectures. It is an interactive environment in which we can simulate MIMD architectures with distributed memory and communication with message passing, as well as MIMD architectures with shared memory, on a uniprocessor UNIX workstation. The computational part of CONLAB is based on the high-performance linear algebra library LAPACK [1], which is portable to many computer architectures and therefore gives the computational kernel of CONLAB good performance on a variety of computers.

CONLAB is an extension of a subset of MATLAB [18] with additional control structures for expressing parallel execution of programs and primitives for message passing, use of shared memory and synchronization. The language used in CONLAB has the abstraction level of the informal algorithm specification languages that many algorithm designers use.

Parallelism is expressed using a control structure, much like a function, a process that can be "called" with a set of arguments. The arguments that can be passed to a process are typically used for initializations. Processes are assigned to virtual processors, which are simulated by time sharing on the UNIX workstation.

Since CONLAB is an interactive environment, the user can define and redefine functions and processes interactively and use them immediately without compiling and linking.

2.1 CONLAB Model for IBM SP

CONLAB has a time model [16, 17] that computes the elapsed time for each virtual processor. The time model is divided into two parts, computations and communications. Simulations of different DMM architectures can be accomplished by varying these models.

Message-passing time is usually modeled as a linear function of message size for two processors that are directly connected. For a more detailed model there are some other components that affect the message-passing time, e.g., a per-hop delay when processors are not directly connected and/or different times for buffered or unbuffered communication. In CONLAB it is possible to model these different types of communication parameters.

To start with we use the following simple model for the communication time,

$$t_{\mathrm{com}}(n) = \gamma n + \delta$$

with a start-up cost, δ, and a per-byte cost, γ, where n is the number of bytes per message. The start-up time latency is related to hardware and software characteristics and can often be considered a constant, i.e., a penalty paid each time the send function is called.

We use the same MPI-based program as in [9] to measure the bandwidth and start-up time. The start-up time should be close to the zero-length communication time and $1/\gamma$ should be the asymptotic bandwidth. The results are shown in Table 1 and indicate that $\delta \approx 32\mu s$ and $\gamma \approx 1/99 s/\text{MB}$. The numbers from Table 1 are also plotted in Figures 1 and 2.

Message size (bytes)	Transfer time (μs)	Bandwidth (MB/s)
0	32.01	0.00
8	37.97	0.21
16	36.48	0.44
32	38.44	0.83
64	37.49	1.71
128	40.47	3.16
256	44.47	5.76
512	52.99	9.66
1024	68.55	14.94
2048	84.46	24.25
4096	115.99	35.31
8192	222.03	36.90
10000	245.99	40.65
100000	1235	80.97
1000000	10349	96.62
10000000	101127	98.89
50000000	503823	99.24

Table 1: Message-passing transfer times and bandwidth on the IBM SP for various message sizes.

The time model for computations in CONLAB is based on the number of floating point operations (flops) a process performs, and the time for one flop (floptime). Since our goal is to have as much level 3 BLAS operations and matrix multiplication is a very common operation in LAPACK and ScaLAPACK, we use the vendor supplied DGEMM – general matrix multiply and add operation – to measure floating point performance, resulting in a "practical" peak performance of 446 Mflops/s ($\approx$ 0.0022 seconds for one flop). If we compare this with the theoretical peak performance of one of the SP nodes, which is 480 Mflops/s, we see that the DGEMM utilizes the machine very well. This is of course a simplified model since it does not take into account the memory hierarchy of modern computers, e.g., registers, level 1 and level 2 cache and so on. The level 1, 2, and 3 BLAS all have different performance, because of the different possibilities to reuse data in the memory hierarchy. Instead of using matrix multiply as basis for measuring the floptime, we could use the performance of the routine for Hessenberg reduction on one processor as "practical"

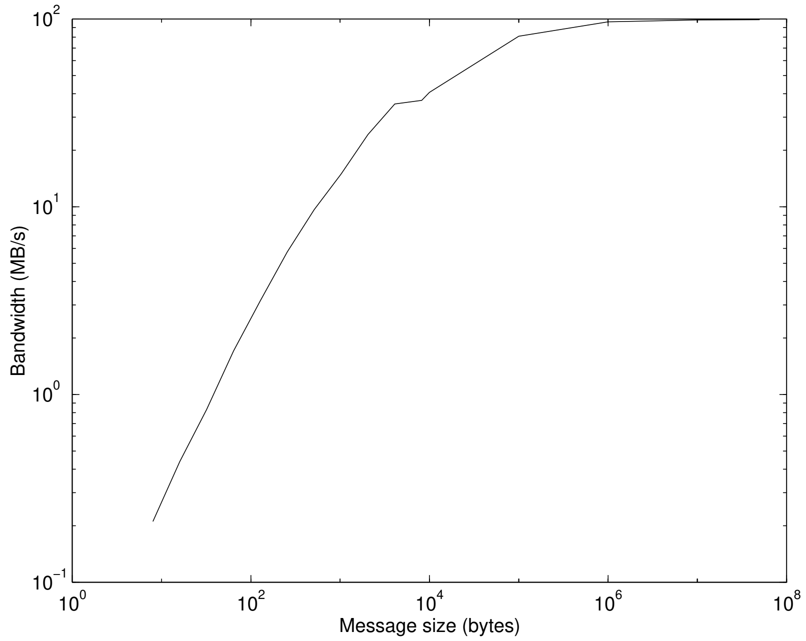

Figure 1: log/log plot of the bandwidth from Table 1.

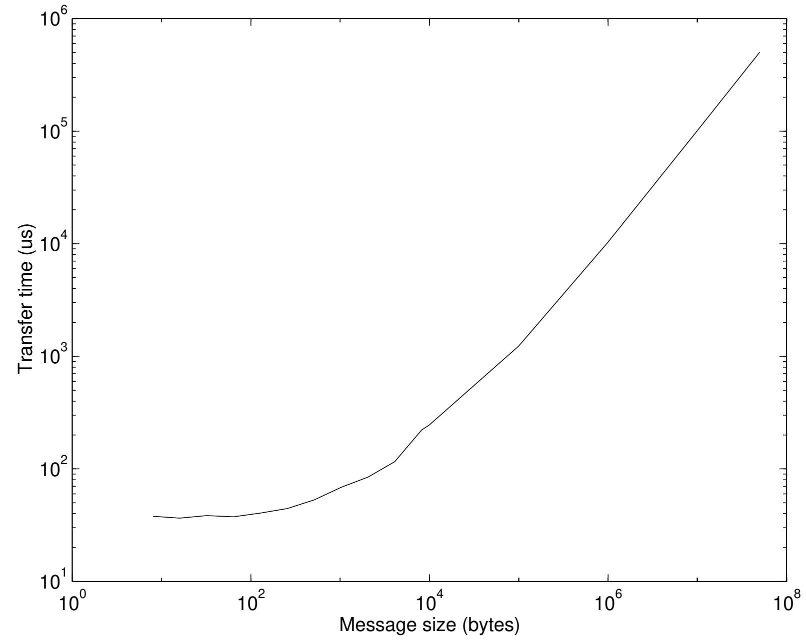

Figure 2: log/log plot of the message transfer times from Table 1.

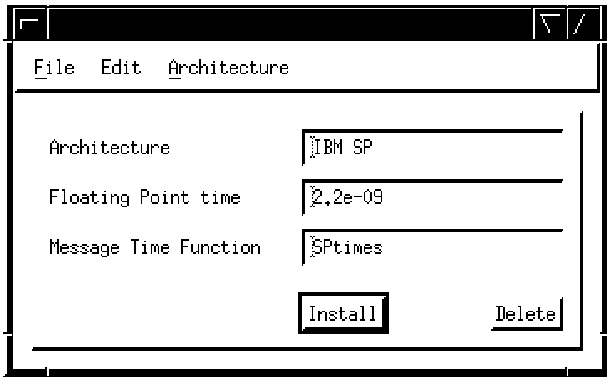

Figure 3: The window for choosing/creating an architecture model in CONLAB.

peak performance for our problem. Tests on the SP show that the maximum speed for this routine is 294 Mflops/s, resulting in 0.0034 seconds for one flop.

All these measured numbers (time for one flop, γ, and δ) are used in the time function that is used by CONLAB to compute the computation and communication times. This CONLAB function with a specified user interface, computes the modeled times of different parts of the communication given the source, destination, and size of the message. Using the **arch** command it is possible to change between different functions that model the architecture characteristics of different systems. Arguments to the **arch** command are the floptime and the name of the communication function. If we omit the arguments a window appears where we can fill in the required information, see Figure 3. We can choose an already existing model via the architecture menu, where all installed models are listed, or we can make a new architecture model by editing a new time function and press the install button. To include new information about communication characteristics, we just have to edit the corresponding communication function. In Figure 4 we show an example of a simple communication function modeling the IBM SP system. We are not using non-blocking communication in our implementation and therefore we ignore all parameters handling the non-blocking communication (i.e., the last six parameters). We are in fact setting them to infinity, just to show that they are not used. To simplify the model even further we ignore the possibility to model unbuffered communication by setting the parameter dl to 0. To be on the safe side we set the parameters for unbuffered send and receive to be equal to the times for buffered send and receive, respectively.

3 Parallel Reduction to Hessenberg Form in ScaLAPACK

The basic approach is to aggregate Householder transformations [13] and to apply them in a blocked fashion [2, 19], thus achieving an algorithm that is rich

```
function [bs, br, us, ut, dl, nbs, nbr, nus, nut, nis, nir, ndl] =
         SPtimes(from, to, type, size)
%
% This function specifies a simplified model for communication costs
% for the IBM SP System.
%
% The parameters have the following meaning:
% The input parameters from, to, type, and size specify the sender
% and receiver nodes and the type and size of the message.
%
% The output parameters bs and br contain the modeled times for
% buffered send and receive, respectively.
% Output parameters us and ut contain the modeled send time and total
% communication time for unbuffered communication.
% Output parameter dl contains the modeled delay time from start of
% sending to start of reception for unbuffered communication.
%
% The output parmeters nbs, nbr, nus, nut and ndl contain the same
% information as the parameters bs, br, us, ut, and dl but for
% non-blocking communication. nis and nir contain the time for
% initializing the non-blocking communication.
%
    gamma  = 1/99e6;
    delta = 32e-6;
    bs = gamma * size + delta;
    br = gamma * size + delta;
    us = bs;
    ut = br;
    dl = 0;
%
% Set all output parmeters for non-blocking communication to infinity.
% (Our way of showing that we are not using non-blocking communication.)
%
    nbs = inf;
    nbr = inf;
    nus = inf;
    nut = inf;
    ndl = inf;
    nir = inf;
    nis = inf;
end
```

Figure 4: CONLAB function for computing communication time components.

in matrix-matrix operations. In ScaLAPACK a parallel version of the algorithm in LAPACK is used [6], and we describe both the sequential and parallel blocked versions briefly [1, 3].

3.1 Hessenberg Reduction

Reduction of an $M \times M$ matrix A to upper Hessenberg form H is done by an orthogonal similarity transformation, $Q^T A Q = H$, where Q is a product of $M-2$ Householder transformations,
$Q = Q^{(1)}Q^{(2)}\cdots Q^{(M-2)}$. The Householder transformation, $Q^{(k)}$, annihilates elements in column k and row k of $A^{(k)}$. Householder transformations are symmetric, and we can write,

$$A^{(k+1)} = Q^{(k)}A^{(k)}Q^{(k)} = Q^{(k)}Q^{(k-1)}\cdots Q^{(1)}AQ^{(1)}Q^{(k-1)}Q^{(k)},$$

where $A^{(1)} = A$ and $A^{(M-1)} = H$. Since each Householder matrix has the form, $Q^{(k)} = I - \tau v v^T$, we can rewrite the equation:

$$A^{(k+1)} = Q^{(k)}A^{(k)}Q^{(k)} = (I-\tau v v^T)A^{(k)}(I-\tau v v^T) = (I-\tau v v^T)\cdot(A^{(k)} - y v^T),$$

where $y = \tau A^{(k)} v$. If we accumulate the Householder transformations such that $V_{k-1} = (v_1, v_2, \cdots, v_k)$, $Y_{k-1} = (y_1, y_2, \cdots, y_k)$, both are $M \times (k-1)$ matrices, and let

$$T_k = \begin{pmatrix} T_{k-1} & -\tau T_{k-1} V_{k-1}^T v \\ 0 & \tau \end{pmatrix}, \qquad (1)$$

then,

$$A^{(k+1)} = (I - V_k T_k^T V_k^T)\cdot(A^{(1)} - Y_k V_k^T),$$

where T_k is a $k \times k$ upper triangular matrix with $T_1 = \tau_1$. This representation (V_k, Y_k, T_k) is a compact version of the block Householder transformation corresponding to $Q^{(k)}Q^{(k-1)}\cdots Q^{(1)}$, see also [2, 19].

3.2 Sequential Block Hessenberg Reduction

Suppose the matrix A is partitioned into block columns (panels), with each panel consisting of n_b consecutive columns of A. In step k of the algorithm, the kth panel is reduced. The Householder vectors for each column are found and are used to update the next column of the panel, but the updating of panels to right is deferred until the reduction of the current panel is completed.

Step k of the LAPACK routine, DGEHRD, consists of three main operations. The subroutines used are mentioned last, inside parenthesis [3].

1. Reduce the kth panel of the matrix and compute V, Y, and T (DLAHRD).
2. Update A with Y, $A \leftarrow A - YV$ (DGEMM).
3. Apply the block Householder vector from the left, $A \leftarrow (I - VT^TV^T)A$ (DLARFB).

In the first step, the Householder vector for each column in the panel is evaluated in turn, and all such vectors computed so far are used to update the next column in the panel. As each column is processed a new column of V, Y, and T is constructed. Each column of a panel is processed and updated by calls to level 2 BLAS routines.

3.3 Parallel Hessenberg Reduction

All global matrices must be distributed on the processor grid prior to calling a ScaLAPACK routine. It is the user's responsibility to perform this data distribution.

ScaLAPACK assumes a one-dimensional or two-dimensional block-cyclic distribution of matrices involved in the computations [3]. In the two-dimensional block-cyclic distribution an $M \times N$ matrix is first partitioned into $MB \times NB$ blocks starting at its upper left corner. These blocks are then distributed in each dimension of the processor grid. Thus, every processor owns a collection of blocks, which are locally and contiguously stored in a two-dimensional "column major" array. The result of the partitioning of a matrix with $M = N = 9$ and $MB = NB = 2$ and mapping these blocks onto a 2×3 processor grid is shown in Figure 5.

The general structure of the parallel Hessenberg reduction algorithm is the same as in the sequential case. Instead of using the single processor routines, their counterparts from ScaLAPACK and PBLAS are used. The routine PDLAHRD is called to reduce each panel, the PBLAS routine PDGEMM is called to apply the block reflector from the left, and PDLARFB applies the block reflector from the right [3].

The inside of PDLAHRD is also very similar to DLAHRD. PDLARFG is called, for the ith column of a panel, to evaluate the Householder vector v_i and the value of τ, see equation (1). Then the Householder vector is distributed over the processes in one column of the processor grid. In order to find the next column of Y and T the next step is to compute Av_i and $V_{i-1}^T v_i$, which is done by PDGEMV. The next step is to apply $(I - V_i T_i^T V_i^T)$ to the column, which is done by a series of calls to PDGEMV for general matrix-vector multiplications, and single processor DTRMV for triangular (T) matrix-vector multiplications.

3.4 Performance Results for PDGEHRD

We have done some tests using the ScaLAPACK routine for reduction of a general matrix to Hessenberg form (PDGEHRD) on an IBM SP system. We use the ScaLAPACK test driver for PDGEHRD, and the results are shown in Figure 6. In the test we have used a block size of 32 since it produces the best results for most of the tested problem sizes (M) and grid sizes ($P \times Q$). We were only able to run problems up to a local matrix size of about 3000 by 3000 on each node. That is why the curves are so short for a small number of nodes in Figure 6.

In the left part of Table 2 the performance results of PDGEHRD are displayed when we keep the local matrix size fixed and increase the global problem

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{28}	a_{29}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{38}	a_{39}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}	a_{58}	a_{59}
a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}	a_{68}	a_{69}
a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}	a_{78}	a_{79}
a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}	a_{87}	a_{88}	a_{89}
a_{91}	a_{92}	a_{93}	a_{94}	a_{95}	a_{96}	a_{97}	a_{98}	a_{99}

9×9 matrix partitioned in 2×2 blocks.

	0				1			2	
0	a_{11}	a_{12}	a_{17}	a_{18}	a_{13}	a_{14}	a_{19}	a_{15}	a_{16}
	a_{21}	a_{22}	a_{27}	a_{28}	a_{23}	a_{24}	a_{29}	a_{25}	a_{26}
	a_{51}	a_{52}	a_{57}	a_{58}	a_{53}	a_{54}	a_{59}	a_{55}	a_{56}
	a_{61}	a_{62}	a_{67}	a_{68}	a_{63}	a_{64}	a_{69}	a_{65}	a_{66}
	a_{91}	a_{92}	a_{97}	a_{98}	a_{93}	a_{94}	a_{99}	a_{95}	a_{96}
1	a_{31}	a_{32}	a_{37}	a_{38}	a_{33}	a_{34}	a_{39}	a_{35}	a_{36}
	a_{41}	a_{42}	a_{47}	a_{48}	a_{43}	a_{44}	a_{49}	a_{45}	a_{46}
	a_{71}	a_{72}	a_{77}	a_{78}	a_{73}	a_{74}	a_{79}	a_{75}	a_{76}
	a_{81}	a_{82}	a_{87}	a_{88}	a_{83}	a_{84}	a_{89}	a_{85}	a_{86}

2×3 processor grid point of view.

Figure 5: A 9×9 matrix partitioned into 2×2 blocks mapped onto a 2×3 processor grid using block-cyclic distribution.

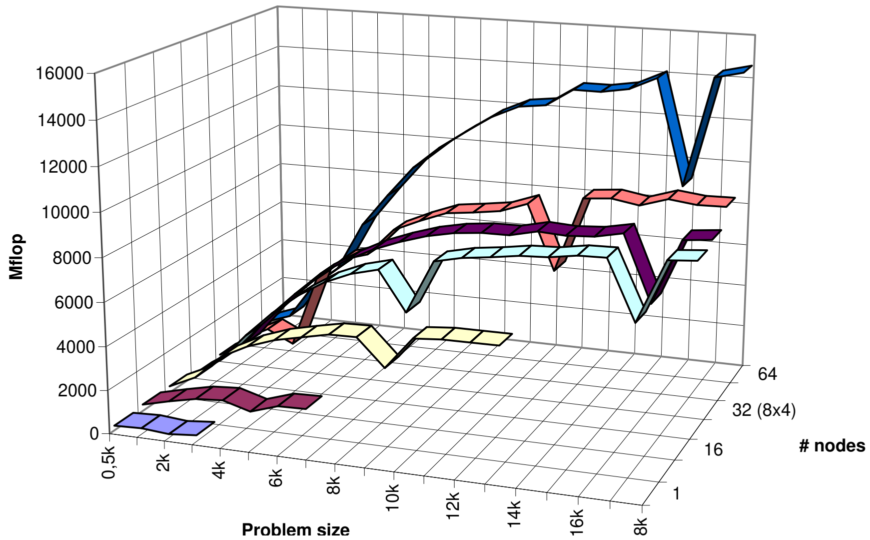

Figure 6: PDGEHRD on IBM SP for different problem and grid sizes.

size with the number of nodes in the processor grid. The parallel speedup S_p is defined as $S_p = T_1/T_p$, where T_1 is the time using sequential code on one processor and T_p is the time using the parallel code on $p = P \cdot Q$ processors. The parallel efficiency E_p is defined as $E_p = S_p/p$. The scaling characteristics of PDGEHRD is very good since the speedup is almost linear with a parallel efficiency of 89% for 36 processors and 86% for 64 processors. The performance figures in this part of Table 2 correspond to the rightmost values of each curve in Figure 6.

In the right part of Table 2 we show what happens if we keep the global problem size fixed and thereby decreasing the local matrix size with increasing number of nodes. The performance of PDGEHRD increases up to some point where the local matrix size gets to small and the communication overhead dominates. As expected for a small number of nodes the speedup is reasonably good.

There is an interesting dip in performance when the local matrix size on each processor is 2048 by 2048 and the block size is 32. This is true for all tested grid

	Constant size of local matrix (3072)				Constant global size (3072)		
nodes	Mflops/s	Global M	S_p	E_p	Mflops/s	Local M	S_p
1	281.67	3072	1.00	1.00	281.67	3072	1.00
4	1079.21	6144	3.83	0.96	993.83	1536	3.53
16	4063.60	12288	14.43	0.90	2676.48	768	9.50
36	9024.57	18432	32.04	0.89	3673.96	512	13.04
64	15516.05	24576	55.09	0.86	3492.58	384	12.40

Table 2: Speedup results for PDGEHRD running on an IBM SP using a fixed local matrix size and a fixed global problem size.

sizes. One probable cause for this is that the local leading dimension (*LLD*) always is the same as the size of the local matrix and that this is interfering with the cache size. The problem can be diminished by changing the *LLD* of the local matrix from 2048 to, for example, 2176 (see Table 3 for more examples of *LLD* changes). Another way to deal with the problem is to use another block size, e.g., 40 instead of 32, but we can not choose the block size arbitrarily. For example, the block size 16 still gives the same poor results as the block size 32 (see Table 3 for more examples of block size changes). If we change the *LLD* or use another block size when necessary, the resulting curve will be smoother with less irregularities (smaller dips), as illustrated in Figure 7. In this test we use $block\ size + 31$ as the *LLD*. In Table 3, we present some more results from using different block sizes. For the block size 32 we also show the impact of changing the *LLD*. Notice, e.g., that adding 3 to the original *LLD* (i.e., 2051 instead of 2048) is not enough to give any real improvement. The table also shows that sometimes changing the *LLD* is best and sometimes changing the block size gives better results.

4 Parallel Reduction to Hessenberg Form in CONLAB

We want to design and implement a non-trivial parallel algorithm in CONLAB and we choose to take an existing algorithm from ScaLAPACK for reducing a general square matrix to Hessenberg form (PDGEHRD) as our starting point. Another objective is to investigate the possibility to simulate the performance of the parallel algorithm on an IBM SP system using the facilities in CONLAB.

4.1 Algorithm Design in CONLAB

We start with the high level subroutine PDGEHRD and all subroutines that it calls. Since the three basic building blocks of ScaLAPACK are the BLACS, PBLAS, and BLAS we have to implement all of these routines that are used by PDGEHRD and every routine used by any implemented routine. We do it this way because we want to have the routines in CONLAB and Fortran as similar as possible in order to be able to make the simulations in CONLAB as

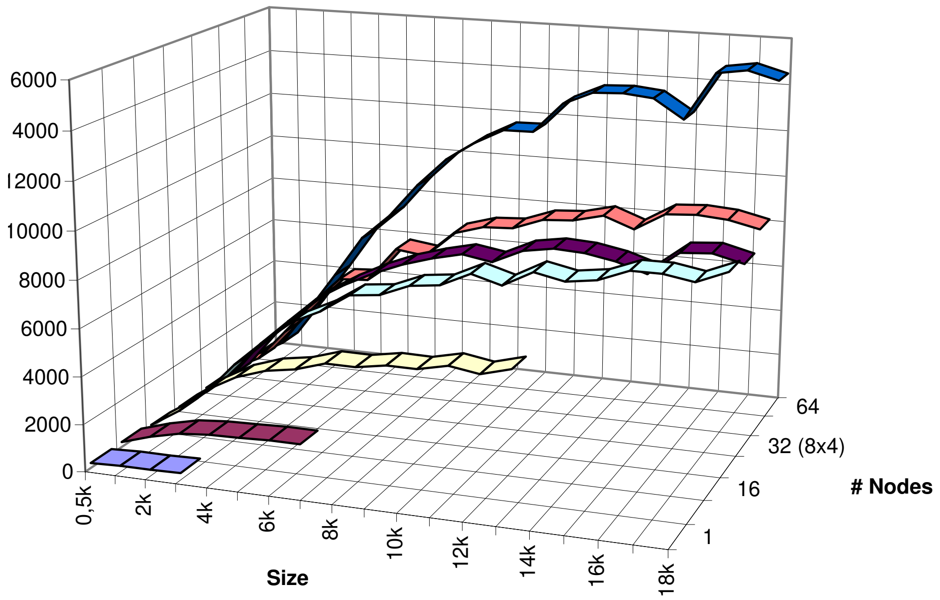

Figure 7: PDGEHRD on IBM SP for different problem and grid sizes with new local leading dimension, $LLD = block\ size + 31$.

Matrix size	P	Q	Block size	LLD	Mflops/s
4096	2	2	16	2048	600.48
4096	2	2	24	2048	1033.02
4096	2	2	32	2048	632.95
4096	2	2	32	2176	**1068.00**
4096	2	2	32	2079	1050.71
4096	2	2	32	2051	635.84
4096	2	2	40	2048	1004.39
8192	4	4	16	2048	2298.20
8192	4	4	24	2048	2449.26
8192	4	4	32	2048	2424.00
8192	4	4	32	2176	3792.55
8192	4	4	32	2079	3954.08
8192	4	4	32	2051	2426.87
8192	4	4	36	2048	2456.93
8192	4	4	40	2048	**4211.73**
8192	4	4	48	2048	2414.42

Table 3: Performance results for PDGEHRD executing on an IBM SP system using different block sizes and local leading dimensions (LLD).

fair as possible. Most subroutines from the BLAS can be expressed directly in CONLAB notation and many of the subroutines from the BLACS can easily be translated into send and receive statements. When triangular matrices are used there is also need for some copying since the communication can only handle contiguously stored blocks.

In the ScaLAPACK implementation the number of rows and columns in a block of the data distribution are chosen to be equal to the block size of the computation. For example, if we choose the computation block size to be 2 for a matrix of size 9×9 then the number of rows and columns of a block in the data distribution will also be 2, and for a 2×3 processor grid we get the distribution shown in Figure 5. This has the nice property that each panel lies in a single column of the process template, and the triangular matrix T is in a single process. Instead of calling the BLAS routines we use their parallel counterparts from PBLAS. This means that almost all communication is hidden inside the PBLAS and are not visible in the top levels of the reduction. In PDGEHRD the routine PDLAHRD is called to reduce each panel. The PBLAS routine PDGEMM is used to apply the block reflector for a panel from the left, and PDLARFB applies the block reflector from the right. In all routines there are calls to information routines to get information about the distributed matrix and the processor grid. In the routines PDLARFG and PDLARFB there are also explicit communication calls, using BLACS, but all other communication are inside PBLAS.

In the rest of this section we describe some of the difficulties experienced during the translation and implementation process.

4.1.1 Ordinary Statements

It is quite simple to translate and implement most of the Fortran code in CONLAB, e.g., assignment statements are very similar, just remember to put an ';' last on each row in order to avoid unnecessary printing of the result. The if-then-else and DO-loop statements are also straightforward to express in CONLAB. Of course we use symbols to represent relational, equality and logical operators (e.g., $<$ instead of `.LT.`) and no labels are used in CONLAB, but the rest is very similar.

4.1.2 Subroutine Calls

We kept (almost) all parameters in subroutine calls, even if they are not used. This makes it easier to translate the calls, and identify the different parameters when trying to debug the CONLAB implementation. The leading dimension of a matrix (LDA) is an example of a parameter that often is not used in the CONLAB implementation (since matrix dimensions are implicit in CONLAB), but in the initial implementation it is passed as an argument to the subroutines.

All subroutines have been implemented as a corresponding CONLAB function. For example, the following subroutine call in Fortran, which returns the number of rows and columns in the processor grid and also the row and column numbers of the calling process inside the processor grid,

```
call blacs_gridinfo(ictxt, nprow, npcol, myrow, mycol),
```

where all arguments but `ictxt` are output parameters, can be expressed in the following CONLAB code:

```
[nprow,npcol,myrow,mycol] = blacs_gridinfo(ictxt);
```

If there are arguments that are both input and output then they must appear on both sides of the assignment, since CONLAB functions do not have reference (output) parameters.

Several subroutine calls are translated directly into CONLAB code without explicit function calls. There is no need to create routines for the single processor BLAS (level 1, 2, and 3) since the functionality of these routines is simple to express in explicit CONLAB code. For example, a call to DGEMM, i.e., general matrix multiply and add, can be expressed directly in CONLAB code as `C = beta * C + alpha * A * B`, where A, B and C are matrices and α and β are scalars.

4.1.3 Goto

There is no goto-statement in CONLAB and this fact could be a problem. However, most goto's are used within different types of loops and can normally be reformulated into an equivalent while- or repeat-until-loop in CONLAB.

4.1.4 Array Indices

Ordinary array references are simple to translate, e.g., the Fortran assignment `nb = desca(NB_)` is written as `nb = desca(NB_);`. But the use of an indexed array (`A(i,j)`) as a reference when calling a subroutine is not that simple in CONLAB since that will just mean the single element `A(i,j)` and we can not reference the other elements of A inside the subroutine. If the index is 1 then we can use the whole row or column, but if it is some other number then we will have to use the size of the matrix in order to get the correct submatrix. The following example forms $C \leftarrow A^T + \beta C$, where C is a transposed condensed block row (column), copied from a scattered block column (row) of A

```
call pbdtr2bt(... A(1, I * NB + 1), lda, beta, C, ...)
```

and is implemented in CONLAB as

```
[m n] = size(A);
C = pbdtr2bt(... A(:, I * NB + 1 : n), lda, beta, C, ...);
```

Another problem in the translation process is that some routines use two-dimensional matrices as if they were one-dimensional arrays (and vice versa, see Section 4.1.5). This means we have to be very careful when translating subroutine calls that make use of this feature. There is also a problem to recognize things that can be done directly (e.g., without a loop construction) in CONLAB but not so in Fortran. As a simple example we can take the code to initialize the vector x, from index N through M, to a constant value Z.

```
      I = N
  10  CONTINUE
      x(I) = Z
      I = I + 1
      IF (I.LE.M) GOTO 10
```

which in CONLAB can be written compactly as `x(N:M) = Z*ones(M-N+1,1);`

4.1.5 Leading Dimensions

Most of the time there is no need to worry about the leading dimension of a matrix since this is implicit in CONLAB. Sometimes there are calls to a subroutine with a one-dimensional array as an argument and the called subroutine assumes that it is a two-dimensional matrix. In cases like this, the leading dimension has to be passed as an argument and also used in all references inside the called subroutine to calculate the correct array indices, which is implicit in Fortran but has to be explicitly expressed in CONLAB. For example, in such a routine we can write `A(i,j) = ...` in Fortran but we have to write `A(j + (i-1)*LD) = ...` in CONLAB.

4.2 Implemented ScaLAPACK Building Blocks

One of the results of this work is that a toolbox of routines has been implemented in CONLAB. These routines are building blocks used in ScaLAPACK and they are very useful for the development and prototyping of new ScaLAPACK-style algorithms. We remark that every subroutine are not fully implemented, because PDGEHRD and the routines necessary to implement it do not use every possible combination of input parameters. In Tables 4 and 5 we show which subroutines we have implemented from ScaLAPACK, BLACS, and PBLAS. A cross in column F means that the implemented CONLAB routine has the full functionality of the corresponding ScaLAPACK routine. Otherwise, a cross is marked in column P (only partial functionality is implemented). Since we only consider the Hessenberg reduction for a double precision matrix, the routines implemented are the routines handling double precision data.

The routines above the line under the ScaLAPACK heading in Table 4 are the computational and auxiliary routines that are needed for the implementation of PDGEHRD. The rest of the ScaLAPACK routines are tools and internal routines.

For the BLACS we have implemented all broadcast routines, two out of four point to point communication routines (omitting the triangular send and receive), and one of three routines for a combined operation and communication (the combine sum operation). The initialization and destruction routines are not needed for CONLAB in this application and are therefore not implemented.

In version 1.4 of ScaLAPACK the PBLAS routines are written in C and most of them are just simplified interfaces to routines from the PB-BLAS library [5]. The PB-BLAS routines have more restrictions on the size and shapes of the matrices involved than the PBLAS, and the use of C is also motivated by the ability to allocate memory dynamically. In Table 5 the implemented PBLAS routines are listed first under the PBLAS heading, and below the horizontal line are the routines from the PB-BLAS library.

More about the different subroutines and their implementation can be found in the Users' Guides (e.g., [3, 10]), and also as comments in the source code. For several algorithms and subroutines there are also technical reports, lawns (LAPACK Working Notes), available, e.g., LAPACK Working Note #100 [7] which describes the proposal for PBLAS.

4.3 Sample CONLAB Code for the Parallel Hessenberg Reduction

In Appendix A we list the CONLAB code for the top level functions of this problem. First there is the simulation function that takes the matrix and grid sizes as parameters and assigns the host program to a virtual processor. Finally it starts the actual simulation. The main steps in the host program are to assign the node program to each virtual processor, distribute the matrix and receive the results. The node programs receive their local part of the matrix from the host, calls the CONLAB function PDGEHRD, and finally sends the results back to the host. The last code is the CONLAB version of PDGEHRD which uses several of the functions from Tables 4 and 5 to implement the Hessenberg reduction.

	F	P	Comment
ScaLAPACK			
pdgehrd	X		Reduces a distributed matrix to Hessenberg form
pdgehd2	X		Reduces a distributed matrix to Hessenberg form (unblocked)
pdlahrd	X		Reduces the first NB columns of a distributed matrix
pdlarf	X		Applies an elementary reflector from left or right
pdlarfb	X		Applies a block reflector from left or right
pdlarfg	X		Generates an elementary reflector of order n
chk1mat	X		Checks that the values associated with one distributed matrix make sense
descset	X		Initializes a descriptor vector
dlamch		X	Determines machine parameters
dlapy2	X		Copies all or part of a matrix
dlaset	X		Initializes a matrix
ilcm	X		Computes the Least Common Multiple
indxg2l	X		Computes the local index of a distributed matrix entry
indxg2p	X		Computes the process coordinate which possesses the entry of a matrix
infog1l	X		Computes the starting local indexes (1-dim)
infog2l	X		Computes the starting local indexes (2-dim)
numroc	X		Computes the NUMber of Rows Or Columns of a distributed matrix
pbchkmat	X		Checks the validity of a descriptor vector for a matrix, and returns info
pbchkvect	X		Checks the validity of a descriptor vector for a vector, and returns info
pdelset	X		Sets the distributed matrix entry A(IA,JA)
pdelset2	X		Returns the distributed matrix entry A(IA,JA) and changes A(IA,JA)
pdtreecomb		X	Parallel combine operation on scalars
ptop	X		Get or set the row, column or all broadcast or combine topologies
ptopget	X		Get the row, column or all broadcast and combine topologies

Table 4: ScaLAPACK routines and building blocks implemented in CONLAB.

	F	P	Comment
BLACS			
blacs_gridinfo	X		Returns information about the processor grid
dgebr2d	X[1]		Broadcast/receive for general double precision arrays
dgebs2d	X[1]		Broadcast/send for general double precision arrays
dgerv2d	X		Globally-blocking point to point general double precision receive
dgesd2d	X		Locally-blocking point-to-point general double precision send
dgsum2d	X[1]		Combine sum operation for double precision rectangular matrices
dtrbr2d	X[1]		Broadcast/receive for trapezoidal double precision arrays
dtrbs2d	X[1]		Broadcast/send for trapezoidal double precision arrays
	[1] Not always using the topology of the mesh.		
PBLAS			
pdgemm		X	Simplified interface (in C) to pbdgemm
pdgemv	X		Simplified interface (in C) to pbdgemv
pdnrm2	X		Returns the 2-norm of a distributed vector
pdscal	X		Multiplies a real distributed vector by a real scalar
pbdgemm		X	Parallel blocked version of DGEMM, $C \leftarrow \alpha AB + \beta C$ (restrictions on A, B, C)
pbdgemv	X		Parallel blocked version of DGEMV, $y \leftarrow \alpha Ax + \beta y$
pbdtran	X		Transposes a column block to row block, or a row block to column block
pbdtrget	X		Forms a row/column block of A from scattered row/column subblocks
pbdtrnv	X		Transposes a column vector to row vector, or a row vector to column vector
pbdtrsrt	X		$T \leftarrow A + \beta T$, T is a sorted condensed block row/col from a block col/row of A
pbdtrst1	X		$y \leftarrow x + \beta y$, y is a sorted condensed row/col vector from a col/row vector of x

Table 5: BLACS and PBLAS routines implemented in CONLAB.

$NB = 4$	$M = 16$	32	48	64	80	96	112	128
2x2	2.01	7.75	17.09	29.78	45.50	63.90	84.59	107.18
4x4	0.85	3.21	7.09	12.48	19.36	27.72	37.51	48.73

Table 6: Some predicted performance results measured in Mflops/s on small problems using CONLAB.

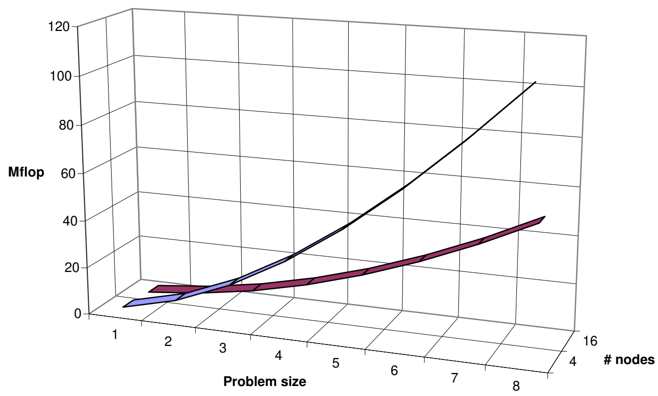

Figure 8: Simulation of an IBM SP running PDGEHRD in CONLAB.

4.4 Performance Analysis of CONLAB Simulations

Earlier studies using CONLAB [16] to develop algorithms on the Intel iPSC/2 hypercube show good correspondence between the predicted times and the real execution times on the parallel machine. There we used simpler algorithms like matrix multiplication and QR factorization on a parallel machine that is much less complicated to model than the IBM SP system.

The current timing model in CONLAB has only one parameter (floptime) that characterizes the cost for doing floating point operations. This simple model is not enough for an accurate modeling of the memory hierarchy of today's advanced computer architecture. However, as we will see the CONLAB timing model is still useful in order to get information about the behaviour of the parallel algorithm.

In Table 6 are some preliminary results from CONLAB, using the timing model we get from the timings in Section 2. Simulations of larger grids than 4×4 demand more memory than we have available, even for small problems. In Figure 8 the numbers from Table 6 are plotted in the same way as the earlier plotted tests from the SP system, and Figure 9 plots the result from test runs on our SP system with the same input as in the CONLAB case. If we compare these plots we see that they look similar and have the same qualitative behavior, but their actual performance differ.

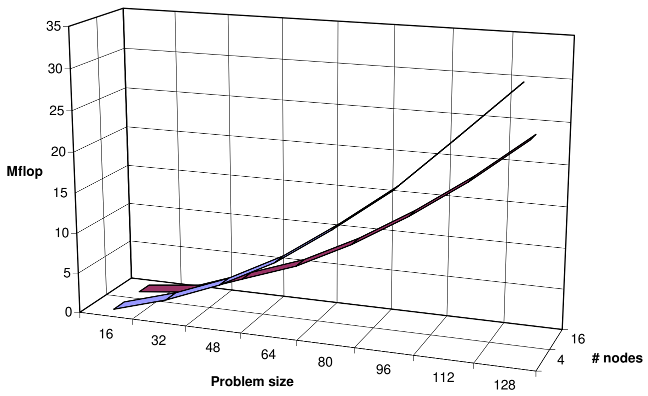

Figure 9: PDGEHRD using small matrices on the IBM SP.

4.4.1 Simulating large problems using down-sizing and scaling

The results reported in Table 6 are for quite small problems. The main reason is that on a uniprocessor, running CONLAB, the memory is not large enough to solve large scale problems. Moreover, the simulation times for realistic problems, simulating many processors each with rather large subproblems on a single processor, would be too long.

Our approach for solving this problem is to "down-size" the problem characteristics (matrix size M and block size NB) so that the simulation time is reasonable and at the same time scale the CONLAB timing model so that the simulation of a down-sized problem mimic the real execution of a large scale problem running on a real parallel machine.

We choose to have the same number of blocks in the down-sized CONLAB problem as in the real large problem. The main advantage is that there will be the same number of communication steps in both problems. The main disadvantage is that there will still be problems that are too large to simulate. In order to be able to solve as large problems as possible, we choose the CONLAB block size to be small ($NB_s = 2$), while keeping the optimal block size ($NB = 32$) when executing the original problem on the IBM SP system. This means that if we, for example, have a problem size M (A is $M \times M$) the down-sized problem will be of size $M_s \times M_s$, where $M_s = NB_s \cdot M/NB$.

The three parameters in the CONLAB timing model are *floptime*, γ and δ (see Section 2). In the following we discuss how these parameters could be scaled or changed in order to be able to emulate the cost (time) for solving the real problem as a down-sized problem in CONLAB.

The cost for reducing an $M \times M$ matrix A to upper Hessenberg form $H = Q^T AQ$ is $10M^3/3$ floating point operations (flops). If the transformation matrix Q is explicitly formed an additional $4M^3/3$ flops are required [13].

By multiplying *floptime* with r^3, where $r = NB/NB_s$, we emulate the

	2×2		4×4	
M	SP	CL	SP	CL
512	306	251	303	314
1024	620	515	899	929
2048	898	725	2037	1963

Table 7: Simulation of large problems executing on an IBM SP in CONLAB using down-sizing and scaling.

arithmetic cost for solving an $M \times M$ problem in CONLAB. Roughly speaking, for each x flops in the down-sized problem there will be $r^3 \cdot x$ flops in the original problem. In our case $r = 16$, i.e., $r^3 = 4096$.

The message passing communication in the parallel algorithm concerns mainly packets containing $NB \times NB$ matrix blocks (submatrices). To compensate for the down-sizing of the matrix block size we scale γ, the per-item communication cost, with $r^2 = 256$. However, there is no need to change δ, the latency for each communication, since the number of sends and receives are roughly the same for the down-sized CONLAB problem and the original problem and the start-up penalty is typically constant and independent of the message size on a real machine.

In Table 7 we present some performance results (measured in Mflops/s) for down-sized problems in CONLAB using the scaled timing model (column CL) and the corresponding real problems of size $M \times M$ executing on the IBM SP system (column SP). The grid sizes used here are 2×2 and 4×4. We conclude that we have a quite good agreement between the simulated performance of the CONLAB algorithm and the real performance of the ScaLAPACK algorithm.

5 Conclusions

We have illustrated how a non-trivial algorithm can be implemented in CONLAB, using the Hessenberg reduction algorithm from ScaLAPACK as our example. One important result of this work is that a toolbox of routines that are used as building blocks in ScaLAPACK are implemented in CONLAB. These CONLAB functions also include several BLACS communication routines and PBLAS computational kernels, which are useful in future algorithm developments.

We have also illustrated how the performance of the ScaLAPACK algorithm running on an IBM SP system can be simulated using the facilities offered by CONLAB. Using the timing model of CONLAB straightforwardly we get information of the general behaviour of the parallel algorithm executing on the machine model defined through a few parameters (cost for one floating point operation, start-up penalty for messages and a per-item cost in the message passing communication). These parameters are of course not enough to model a real parallel machine accurately. Especially, the timing model must be enlarged to cope with a memory hierarchy of today's RISC-type processors. This is a non-trivial task in its own but we point out that it is possible to do this within CONLAB, e.g., by defining your own timing function.

Since the simulations performed in CONLAB runs on a Unix workstation there are several limitations on, e.g., the problem sizes and the processor grid sizes to be used in the simulations. The main problem is the lack of memory and in order to be able to solve realistic problems in CONLAB we propose and illustrate a down-sizing and scaling approach for the emulation.

Acknowledgement

I would like to thank my supervisor professor Bo Kågström for valuable discussions during the work and preparation of this paper. Moreover, I acknowledge the High Performance Computing Center North (HPC2N) for making the IBM SP system available for the computational experiments.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Release 2.0.* SIAM Press, Philadelphia, PA, 1995.

[2] C. Bischof and C. Van Loan. The WY representation for products of Householder matrices. *SIAM J. Sci. Stat. Comp.*, 8:s2–s13, 1987.

[3] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J.J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D.W. Walker, and R.C. Whaley. *ScaLAPACK Users' Guide.* SIAM, Philadelphia, PA, 1997.

[4] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. LAPACK Working Note 95, Department of Computer Science, University of Tennessee, Knoxville, TN, August 1995.

[5] J. Choi, J. Dongarra, and D. Walker. PB-BLAS: A Set of Parallel Block Basic Linear Algebra Subroutines. *Concurrency: Practice and Experience*, 8(7):517–535, 1996.

[6] J. Choi, J. Dongarra, and D. W. Walker. The Design of a Parallel Dense Linear Algebra Software Library: Reduction to Hessenberg, Tridiagonal, and Bidiagonal Form. *Numerical Algorithms*, 10:379–399, 1995.

[7] J. Choi, J. J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R.C. Whaley. A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. Technical Report CS-95-292, Department of Computer Science, University

of Tennessee, Knoxville, TN, May 1995. (Also LAPACK Working Note #100).

[8] J. Choi, J.J. Dongarra, R. Pozo, and D.W. Walker. ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. LAPACK Working Note 55, Department of Computer Science, University of Tennessee, Knoxville, TN, November 1992.

[9] J. Dongarra and T. Dunigan. Message-passing Performance of Various Computers. *Concurrency: Practice and Experience*, 9(10):915–926, 1997.

[10] J. J. Dongarra and R.C. Whaley. A User's Guide to the BLACS v1.1. Technical Report CS-95-281, Department of Computer Science, University of Tennessee, Knoxville, TN, 1995. (Also LAPACK Working Note #94).

[11] P. Drakenberg, P. Jacobson, and B. Kågström. A CONLAB Compiler for a Distributed Memory Multicomputer. In R. Sincovec et al, editor, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, volume Vol. II, pages 814–821. SIAM Publications, Philadelphia, 1993.

[12] J. Eriksson, P. Jacobson, B. Kågström, and E. Lindström. The CONLAB Environment: Algorithm Design for and Simulation of MIMD Architectures. In J. Dongarra, P. Messina, D. Sorensen, and R. Voigt, editors, *Parallel Processing for Scientific Computing*, pages 406–412. SIAM Publications, Philadelphia, 1990.

[13] G. Golub and C. Van Loan. *Matrix Computations*. Second Edition. Johns Hopkins University Press, Baltimore, MD, 1989.

[14] P. Jacobson. The CONLAB Environment. Technical Report UMINF–173.90, Institute of Information Processing, University of Umeå, S–901 87 Umeå, SWEDEN, May 1990.

[15] P. Jacobson. The CONLAB Simulator – Execution and Scheduling of Parallel Computations. Technical Report UMINF–92.08, Institute of Information Processing, University of Umeå, S–901 87 Umeå, Sweden, May 1992.

[16] P. Jacobson, B. Kågström, and M. Rännar. Algorithm Development for Distributed Memory Multicomputers Using CONLAB. *Scientific Programming*, 1:185–203, 1992.

[17] H. Magnusson and B. Torkelsson. Software Development Environments for Parallel Computer Systems (in Swedish). *Master Thesis*, Report UMNAD-160.96, Department of Computing Science, Umeå University, S-901 87 Umeå, Sweden, 1996.

[18] C. B. Moler, J. Little, and S. Bangert. *PRO-MATLAB User's Guide*. The Mathworks Inc., 1987.

[19] R. Schreiber and C. Van Loan. A Storage Efficient WY Representation for Products of Householder Transformations. *SIAM J. Sci. Stat. Comp.*, 10:53–57, 1989.

A Sample CONLAB Code

Simulation Function

```
function dummy = sim(A, blksize, P, Q)
%
%   Simulation function for parallel Hessenberg
%   reduction (pdgehrd)
%   A = matrix to be reduced, blksize = block size,
%   P x Q =  processor grid
%   Uses processes hess_host and hess_node
#include    "blacs.h"
%   Start the host process on virtual processor P*Q
%   giving it process id HOST
%
    assign(hess_host(A, blksize, P, Q), P*Q, HOST);
    simulate;
end
```

Host Program

```
process hess_host(A, nb, P, Q)
%
% Host program for Hessenberg reduction (pdgehrd)
% nb = blocksize, P x Q = processor grid
%
#include    "blacs.h"
%
% Test if P and Q are valid
%
if P ~= PROCINROW | Q ~= PROCINCOL
    disp('ERROR using hess_host:');
    disp('    The processor grid is not valid);
    disp('    trying to use', P, ' x', Q);
    disp('    when the grid defined in blacs.h is', ...
          PROCINROW, ' x', PROCINCOL);
    return;
end

assign(hess_node(nb, P, Q),0:P*Q-1);
%
% Distribute A
%
for i = 0 : P*Q-1
    localmatrix = scatter(i, A, nb, P, Q);
    send(localmatrix, i, SCATTERID);
end
%
% Receive results
%
[m n] = size(A);
```

```
H = gather(zeros(m, n), nb, P, Q);
%
% Receive TAU
%
tau = zeros(m, 1);
for i = 1 : Q
    [t from] = receive(default, TAUID);
    x = 1;
    for j = 1 + from*nb : nb*Q : m
        tau(j : j+nb-1) = t(x : x+nb-1);
        x = x + nb;
    end
end
%
% Compute simulated parallel time
%
maxtime = 0;
for i = 0 : P*Q-1
    [tid from] = receive(default, TIMERID);
    maxtime = max([tid maxtime]);
end

% HRD requires 4/3*(IHI-ILO)^3 + 2*IHI*(IHI-ILO)^2 flops
[m n] = size(A);
disp(' ');
disp('Scaled problem:');
n = n * 16;                             % nb / nb_s = 16
flps = (4/3*(n-1) + 2*n) * (n-1)*(n-1);
disp('Algorithmic number of flops = ', flps);
disp(' ');
disp('flops per second = ', flps / maxtime);

end
```

Node Program

```
process hess_node(nb, P, Q)
%
% Node program for Hessenberg reduction (pdgehrd)
%
#include    "blacs.h"

    me = getpid;
%
%   Receive my part of A, store in A
%
    A = receive(HOST, SCATTERID);
    [m n] = size(A);
%
%   Start the timer and call pdgehrd
%
```

```
    descA = descset(m*P, n*Q, nb, nb, 0, 0, 0, m);

    sync([0:P*Q-1]);
    time = timer; flps = flops; % time this process

    [A tau] = pdgehrd(m*P, 1, m*P, A, 1, 1, descA);

    total = timer - time;
%
%   Send the resulting A to the host
%
    send(A, HOST, GATHERID);
%
%   Only the first row of processors sends tau
%
    if me < PROCINROW
        send(tau, HOST, TAUID);
    end
%
%   Send timing to host
%
    send(total, HOST, TIMERID);

end % process hess_node
```

PDGEHRD

```
function [A tau] = pdgehrd(N, ilo, ihi, A, ia, ja, descA)
%
%  PDGEHRD reduces a real general distributed matrix sub( A )
%  to upper Hessenberg form H by an orthogonal similarity transforma-
%  tion:  Q' * sub( A ) * Q = H, where
%  sub( A ) = A(IA+N-1:IA+N-1,JA+N-1:JA+N-1).
%
#include    "blacs.h"

%   Test of parameters
    nb = descA(NB_);
    ictxt = descA(CTXT_);
    [nprow,npcol,myrow,mycol] = blacs_gridinfo(ictxt);
    iroffa = mod(ia-1, nb);
    [iia,jja,iarow,iacol] = infog2l(ia, ja, descA, nprow, npcol, ...
                                    myrow, mycol);
    ihip = numroc(ihi+iroffa, nb, myrow, iarow, nprow);
    ioff = mod(ia+ilo-2, nb);
    ilrow = indxg2p(ia+ilo-1, nb, myrow, descA(RSRC_), nprow);
    ihlp = numroc(ihi-ilo+ioff+1, nb, myrow, ilrow, nprow);
    ilcol = indxg2p(ja+ilo-1, nb, mycol, descA(CSRC_), npcol);
    inlq = numroc(N-ilo+ioff+1, nb, mycol, ilcol, npcol);
    lwmin = nb * (nb + max(ihip+1, ihlp+inlq));
    Wipt = zeros(descA(NB_));
```

```
    Wipt(1, 1) = lwmin;
%
%   Set elements of tau to zero.
%
    tau = zeros(N/2,1);
    nq = numroc(ja+N-2, nb, mycol, descA(CSRC_), npcol);
    [jj,imcol] = infog1l(ja+ilo-2, nb, npcol, mycol, descA(CSRC_));
    tau(jja : min(jj, nq)) = zeros(1:min(jj, nq)-jja+1);
    [jj,imcol] = infog1l(ja+ihi-1, nb, npcol, mycol, descA(CSRC_));
    tau(jja : nq) = zeros(1:nq-jja+1);
%
%   Quick return if possible
%
    if ihi - ilo <= 0
        return;
    end
    ipt = 1;
    ipy = ipt + nb * nb;
    ipw = ipy + ihip * nb;
    descY = descset(ihi+iroffa, nb, nb, nb, iarow, ilcol, ictxt, ...
                    max(1,ihip));
    k = ilo;
    ib = nb - ioff;
    jy = ioff + 1;
%
%   Loop over remaining block of columns
%
    for l = 1 : nb : ihi-ilo+ioff-nb
        i = ia + k - 1;
        j = ja + k - 1;
%
%       Reduce columns j : j+ib-1 to Hessenberg form
%
        [A,tau,Wipt,Wipy] = pdlahrd(ihi, k, ib, A, ia, j, descA, ...
                                    tau, Wipt, 1, jy, descY);
%
%       Apply the block reflector ... from the right ...
%
        [ei,A] = pdelset2(A, i+ib, j+ib-1, descA, 1.0);
        A = pdgemm('N', 'T', ihi, ihi-k-ib+1, ib, -1.0, Wipy, ...
                  1, jy, descY, A, i+ib, j, descA, 1.0, A,   ...
                  ia, j+ib, descA);
        A = pdelset(A, i+ib, j+ib-1, descA, ei);
%
%       Apply the block reflector ... from the left
%
        A = pdlarfb('L', 'T', 'F', 'C', ihi-k, N-k-ib+1, ib, ...
                   A, i+1, j, descA, Wipt, A, i+1, j+ib, descA);
        k = k + ib;
        ib = nb;
        jy = 1;
```

```
        descY(CSRC_) = mod(descY(CSRC_) + 1, npcol);
    end
%
%   Use unblocked code to reduce the rest of the matrix
%
    [A tau] = pdgehd2(N, k, ihi, A, ia, ja, descA, tau);
end
```