

# Sparse Approximate Inverses in Matlab \*P

Gurpreetsingh Sachdev, Stephanie Taylor

June 7, 2004

## 1 Introduction

We consider the linear system of equations

$$Ax = b \tag{1}$$

where  $A$  is a large, sparse, possibly asymmetric matrix. If we can construct a matrix  $M$  that approximates the inverse of  $A$ , then we can use an iterative method (such as GMRES or BICGSTAB) to solve the preconditioned system

$$MAx = Mb \tag{2}$$

Incomplete LU factors are widely used to speed up the convergence by reducing the number of iterations of iterative methods. However, ILU factorization does not lend itself well to parallel systems. Its computation requires triangular solves, which are sequential in nature.

In this paper, we focus on a preconditioner that can be constructed in parallel. The sparse approximate inverse  $M$  of a matrix  $A$  is a sparse matrix such that  $M \approx A^{-1}$  and the number of non-zeros in  $M$  is on the order of the number of non-zeros in  $A$ .

In a previous paper, [3], we discussed our implementation of the sparse approximate inverse algorithm (from [1]) in Matlab \*P. Matlab \*P is an extension of Matlab that allows the user to perform parallel operations. The approach used in [3] was embarrassingly parallel, but was constrained by its large memory requirements.

In this paper, we discuss a more scalable approach - one that reduces the space requirements on each processor. To do this, we must introduce interprocess communication. Using a message-passing model, we developed a basic version of SPAI. This provided us the opportunity to take a detailed look into the viability of the message-passing model as a communication mechanism for the parallel SPAI algorithm.

## 2 SPAI

SPAI [1] attempts to find the  $M$  that minimizes the Frobenius norm of the residual matrix  $AM - I$ . Since

$$\|AM - I\|_F^2 = \sum_{k=1}^n \|(AM - I)e_k\|_2^2, \tag{3}$$

the solution of (3) separates into  $n$  independent least-squares problems

$$\min_{m_k} \|Am_k - e_k\|_2, \quad k = 1, \dots, n, \quad (4)$$

where  $e_k$  is the  $k^{\text{th}}$  column of the  $n \times n$  identity matrix.

## 2.1 Sparsity Patterns

The inverse of a sparse matrix tends to have significantly more non-zeros than the original matrix. Since we want  $M$  to be sparse (and therefore inexpensive to store and use), some control must be exerted over the fill-in. One approach would be to choose a sparsity pattern for  $M$  a priori, and compute only those values of  $M$  that appear in the pattern. Another approach would be to automatically adapt the sparsity pattern of  $M$ , based on the current value of the residual.

The implementation in [3] allows the user to choose between a fixed sparsity pattern or an adaptive sparsity pattern. Using a fixed-pattern method requires fewer computations (because each column is computed exactly once). On the other hand, using the adaptive method may find a better approximation to  $A^{-1}$  with less fill-in, but it takes longer to compute (because there are multiple computations per column).

The scalable implementation discussed in this paper allows for a fixed pattern only. In our investigation into the communication design, we chose to focus on this simplified version of the problem. However, we believe our design is flexible enough that reinserting the adaptive mode will be relatively straight-forward. We were careful not to make the assumption that the communication would follow a predictable pattern.

### 2.1.1 Fixed

For simple problems, a common fixed sparsity pattern is that of  $A$  itself. For more complicated problems, it may be more effective to use the pattern of  $A^k$  where  $k$  is a small integer  $\geq 2$ , an approach justified by the Neumann series expansion of  $A^{-1}$  [2].

### 2.1.2 Adaptive

In [1], Grote and Huckle attempt to “capture the sparsity pattern of the main entries of  $A^{-1}$  automatically, yet at a reasonable cost.” Their algorithm starts with a minimal sparsity pattern, solves for  $m_k$ , and computes the current residual

$$r = Am_k - e_k. \quad (5)$$

If the residual is not small enough, the algorithm chooses new non-zero positions, updates  $m_k$ , and updates the residual. This process is repeated until the residual is small enough or  $m_k$  becomes too dense. In our implementation, the original sparsity pattern is diagonal.

## 2.2 Algorithm

We will provide the overview of the SPAI algorithm [1] as summarized in [2] and highlight the features most relevant to our implementation, but we refer the reader to [1] for further details.

The algorithm for the fixed mode is captured in the first two steps.

For every column  $m_k$  of  $M$ :

1. Choose an initial sparsity pattern  $\mathcal{J}$  (i.e. the set of indices where  $m_k$  will have nonzero entries).
2. Compute the row indices  $I$  of the corresponding nonzero entries and the QR-decomposition of  $A(\mathcal{I}, \mathcal{J})$ . Then, compute the solution  $m_k$  of the least squares problem (4) and its residual (5).

**The remaining steps are used in the adaptive mode only:**

While  $\|r\|_2 > \epsilon$ :

3. Set  $\mathcal{L}$  equal to the set of indices  $l$  for which  $r(l) \neq 0$ .
4. Set  $\tilde{\mathcal{J}}$  equal to the set of all new column indices of  $A$  that appear in all  $\mathcal{L}$  rows but not in  $\mathcal{J}$ .
5. For each  $j \in \tilde{\mathcal{J}}$  compute the norm of the new residual using the formula

$$\rho_j^2 = \|r\|_2^2 - \frac{(r^T Ae_j)^2}{\|Ae_j\|_2^2} \quad (6)$$

and remove from  $\tilde{\mathcal{J}}$  all but the most profitable indices.

6. Determine the new indices  $\tilde{\mathcal{I}}$  and update the QR-decomposition. Then, solve the new least squares problem, compute the new residual, and set  $\mathcal{I} = \mathcal{I} \cup \tilde{\mathcal{I}}$  and  $\mathcal{J} = \mathcal{J} \cup \tilde{\mathcal{J}}$ .

## 3 Embarrassingly Parallel Approach

### 3.1 Matlab \*P

Matlab \*P is a parallel version of Matlab that allows the user to write code in Matlab's language, but adds features to allow data parallelism and operational parallelism [8]. We use both aspects:

1. Data Parallelism - Data objects can be distributed across processors. In our case, the most obvious example is of  $M$  itself - each processor creates its own section of  $M$  and when *SPAI* returns,  $M$ 's data remains distributed.
2. Operational Parallelism - The operator `mm` (multi-matlab mode) allows users to run the same code on different processors. The data passed to that code can be distributed (in which case, each processor gets only its share), or not distributed (in which case, each processor gets an entire copy).

Since Matlab is a high-level language with built-in mathematical data types and operations, development of computational software is faster than it would be in a lower level language. The same holds for Matlab \*P.

### 3.2 Implementation

We use `mm` to invoke the code (`distribSpai`) on each processor. `distribSpai` generates a subset of  $M$  on its processor, and when `mm` returns, it collects the various columns into a distributed matrix. One limiting factor is that `mm` will accept only dense matrices. Consequently, we send and return dense matrices.

In the embarrassingly parallel approach, each processors gets an entire copy of  $A$ . In order to perform the least-squares solve for  $m_j$ , we need the subset of  $A$ ,  $A(\mathcal{I}, \mathcal{J})$ . The sets  $\mathcal{I}$  and  $\mathcal{J}$  can contain indices from anywhere in the matrix, so we must allow each processor access to all of  $A$ . The simplest way to allow access is to place an entire copy of  $A$  on each processor.

Here is a code snippet - we send an entire, dense copy of  $A$  to each processor:

```
% Sending A as a dense matrix
M = mm('distribSpai',full(A),colIdx,pattern,epsilon, ...
      maxPerIteration,maxPerColumn);
```

This approach is simple, and effective (as shown in [1] and [3]). However, as  $A$  grows in size, it will be less and less desirable to allow one copy on each processor. SPAI is intended to serve a preconditioner for large systems of linear equations that are being solved in parallel. The two major reasons to switch from sequential to parallel solvers are

1. Space Constraints
2. Time Constraints

Usually, the two work together. If we want to ensure that SPAI will be effective as matrices grow in size, we must address the space constraints issue. We must move to an approach in which  $A$  is distributed.

## 4 Scalable Approach

In the more scalable approach,  $A$  is distributed across the processors. However the SPAI algorithm requires that each processor have access to all of  $A$ . To enable this access, it is imperative to introduce communication among the processors. This, in turn, forces a requirement for `mm` - it must provide some form of interprocess communication.

### 4.1 Previous Work

In [5], Stephen T. Barnard, Luis M. Bernardo, and Horst D. Simon discuss an MPI implementation of SPAI that allows  $A$  to remain distributed. They conclude that SPAI scales well and serves as

an effective preconditioner for large systems of linear equations.

In [6], Stephen T. Barnard and Marcus J. Grote present a similar implementation. In this paper, they introduce a Block version of SPAI - one that exploits the block-format present in many sparse systems of equations (e.g. partial differential equations such as found in fluid dynamics). They, too, use MPI [7] and have promising results.

Based on these findings, we set about to implement our own version. We decided to continue coding in Matlab \*P, but have left open the option of importing their code as a package, and writing the "glue" code necessary to convert our data types to theirs. In either case, we must write the fixed mode version ourselves, because they have only the adaptive mode.

## 4.2 Communication Infrastructure in Matlab \*P

Until May 2003, `mm` had no mechanism for interprocess communication. Max Goldman and Da Guo made a significant contribution to Matlab \*P when they implemented an MPI-like interface to enable message-passing in `mm` calls [4].

Goldman and Guo's implementation provides a subset of MPI that is sufficient for our needs - it allows us to request remote parts of  $A$  from other processors. The extension of `mm` that provides MPI support is called `mmpi`.

In order to overlap computational work with communication, it is important to use asynchronous sends and receives. The four functions enabling such communication include `isend`, `irecv`, `testReq`, and `waitReq`.

The Java calls can be made directly in Matlab \*P code, and have the format:

```
public int isend(final double[] buffer, final int count, final int dest, final int tag)

public int irecv(final int count, final int source, final int tag)

public boolean testReq(int ident)

public double[] waitReq(int ident, boolean forRecv)
```

The integers returned by `isend` and `irecv` serve as identifiers for their respective messages, and are passed to calls to `testReq` (which returns `true` if the message send/receive has completed) and `waitReq` (which waits until the message send/receive has completed, and returns any data that is being received).

These particular functions had not been exercised strenuously before we wrote the SPAI code. We found there was little provision for multiple messages to be sent "at the same time" between the same destination-source processor pair. If multiple messages were sent with differing tags, then the `irecv` code depended upon the caller knowing the order in which the messages were going to arrive. If `irecv` was called and the tag of the incoming message was not the one it was looking for, then it ignored the message (and made it inaccessible to all other code). We fixed the problem and all messages are now received, regardless of order.

There were a few synchronization issues and problems reusing the port, but those have been fixed.

## 4.3 Design

The basic design issues outlined in [5] include:

1. One-Sided Communication
2. Latency-Hiding
3. Load Balancing

We will discuss how our design addresses those issues.

### 4.3.1 One-Sided Communication

Different processors need to be able to access remote parts of  $A$ . MPI does not allow for one-sided communication (in which a processor can make a request and receive data without another processor explicitly waiting for the request). So, we have developed a communication scheme that calls for every processor to be expecting a data-request message from every other processor. We use two message tags - data-request and data-receive. The first identifies a message as a request for a column. The second identifies a message as an entire column of data. Whenever a processor is running communication-handling code, it checks to see if it has completely received a request from another processor. Then, it asynchronously fills the request, sending data using the data-receive tag.

Each processor needs to be able to respond to requests when

1. it is doing computation,
2. it is waiting for its own request to be fulfilled, or
3. it has completed its computation.

### 4.3.2 Latency-Hiding

Efficient parallel code overlaps computation time with communication time insofar as possible. Our code is structured so that it explicitly handles communication once per iteration. At the beginning of each iteration, a processor determines which columns it needs and sends requests for them. While it is waiting for those requests to be filled, it fills any requests sent to it. When it has all the data it needs for that iteration, and when there are no requests queued up at its door, it returns to computation.

This means we have two areas of overlap:

1. Request messages can arrive while a processor is in its computation phase.
2. Request messages can be filled for other processors while a processor is waiting for its requests to be filled.

As the design currently stands, there is no overlap of data-receive messages and computation. One reason for this is that when we implement the adaptive mode, we will not know what columns we need until we are ready to perform computation with their data. Another reason is that the bookkeeping required to send the requests for a column or two ahead is much more difficult.

### 4.3.3 Load Balancing

To balance the computation time of the processors, we would need to assign different numbers of columns of  $M$  to each processor. `mm` has a restriction that each processor must return exactly the same size matrix. This works well with the current setup, in which each processor simply computes  $N/NP$  columns of  $M$  where  $M$  is  $N \times N$  and  $NP$  is the number of processors. Each processor returns an  $N \times N/NP$  submatrix.

As the design of `mm` stands, implementing load-balancing would be impractical - it would require even more bookkeeping code and it would take up too much space. Since each return matrix must be the same size as all the others, and each return matrix must be dense, we must allocate for each matrix the amount of space for the largest matrix any processor might return. The maximum size of any return submatrix is not predictable, we must require each submatrix be  $N \times N$  (with an additional row or column to hold bookkeeping information). But if we are going to have full  $N \times N$  matrices on each processor, then it will behoove us to return to the embarrassingly parallel approach and place all of  $A$  on each processor.

## 4.4 Implementation

### 4.4.1 Basic Structure

Here is the basic structure of `distribSpai`. Variable names are based on the symbols used in the algorithm outlined above.

First, MPI is initialized, and the processor sets itself up to receive requests from other processors.

```
% M is the submatrix of M that we must compute.
% P is the pattern for M
% fullA is the dense version of this processors part of A.
% colIdx is the list of column numbers we need to compute.
% colCnt is the number of nonzeros in each column of A (the whole of A).
function M = distribSpai(P,fullA,colIdx,colCnt)

    import mpi.*;

    % This is jsut like MPI_Init.
    MPI.init;
    rank = MPI.COMM_WORLD.rank;
    nproc = MPI.COMM_WORLD.size;

    M = [];
    A = sparse(fullA);
```

```

% Deal with parameters...
...

% Set up receive buffer for requests from other
% processors.
req_wait_list = zeros(nproc,1);
dataRecvTag = 31;
dataRequestTag = 30;
for i = 1 : nproc,
    if (i-1 ~= rank)
        req_wait_list(i) = MPI.COMM_WORLD.irecv(2,i-1,dataRequestTag);
    else
        req_wait_list(i) = -1;
    end;
end;

```

The next section of code contains the main loop. At each iteration of the loop, one column of  $M$  is computed.

```

% Loop creating each column of M.
for k = colIdx(1) : colIdx(end),
    % Find the non-zero pattern of the kth column of M
    Pj = P(k-colIdx(1)+1,:);

    % Consider the subset of A that contains the columns matching
    % the positions of non-zeros in mk, leaving only the columns J.
    % Then remove any rows that have all zeros, leaving only the rows I.
    % This is the subset of A that we are going to work with.

    % Handle communication for a bit.
    % - req_wait_list will be updated to reflect any new requests.
    % - newCols will contain the data of the non-zero columns of
    %   A(:,J) that are not local to this processor.
    % - Jremote is the subset of J that identifies columns on other
    %   processors
    [newCols req_wait_list Jremote] = handleCommunication(A,colIdx(1),...
                                                         colIdx(end),true,J,...
                                                         colCnt,req_wait_list);

    % Now, compute Aij(I,J) to be the A(I,J) mentioned in the
    % algorithm (in the algorithm "A" is all of A).
    Jloc = setdiff(J,Jremote);
    Aij = newCols;
    for i = 1 : length(Jloc),
        Aij(:,Jloc(i)) = A(:,Jloc(i)-colIdx(1)+1);
    end;

```

```

[I, tmpJ, Ai] = find(Aij(:,J));
% Eliminate repeat entries.
I = unique(I);

% Solve the least-squares problem Aij(I,J)*mhat = ehat
% to find the value of mk with its original sparsity
% pattern.
% ehat is the jth column of the identity matrix, subsetted
% to be the same size as A(I,J).
ehat = spalloc(n,1,1);
ehat(k) = 1;
ehat = ehat(I,:);
% Solve the least squares problem using QR decomposition.
[Q R] = qr(Aij(I,J));
mhat = R \ (Q'*ehat);

% Update the current column of M
% (well, do that but store it as a row instead).
M(k-colIdx(1)+1,sortedJ) = mhat';
end; % end loop over columns

```

Once this processor has finished computing its columns of  $M$ , it has finished requesting data from other processors. However, it must be available to fill data requests until other processors have finished as well. This processor will send a request for column  $-1$  to every other processor to indicate it will not send any more requests. When it has received  $-1$  requests from all other processors, it will return.

```

% Let other processors know we have finished
J = [];
finish_wait_list = (-1) * ones(nproc,1);
buff = [-1 -1];
for i = 1 : nproc,
    if (i-1 ~= rank)
        finish_wait_list(i) = MPI.COMM_WORLD.isend(buff,2,i-1,dataRequestTag);
        MPI.COMM_WORLD.waitReq(finish_wait_list(i),false);
        finish_wait_list(i) = -1;
    end;
end;
% Now, wait until we have all -1 messages from
% the rest of the processors.

% Fulfill requests for other processors.
while (sum(req_wait_list) ~= (-1)*nproc),
    [nn req_wait_list jj ] = handleCommunication(A,colIdx(1),colIdx(end),...
                                                true,J,...
                                                colCnt,req_wait_list,fid);
end;

```

```

for i=1:nproc,
    if (finish_wait_list(i) ~= -1)
        if (MPI.COMM_WORLD.testReq(finish_wait_list(i)))
            finish_wait_list(i) = -1;
        end;
    end;
end;

% This is just like MPI finalize
MPI.fnlz;

```

#### 4.4.2 Communication Handling Subroutine

The code in this routine is mostly bookkeeping code - it keeps track of what messages this processor is expecting from other processors and whether or not they have arrived.

```

messThere = true;
% While we are still expecting data (recv_data_wait_list is not
% all -1's), or we are still receiving requests, keep looping.
while (messThere || sum(recv_data_wait_list) ~= (-1)*nproc)
    messThere = false;
    % Check to see if we have received data from any other
    % processors. If so, then we may need to send another
    % request to that processor (for a different column).
    for i = 1 : nproc,
        if (recv_data_wait_list(i) ~= -1)
            if (MPI.COMM_WORLD.testReq(recv_data_wait_list(i)))
                % We have received data. Put it into a set we are using
                % as a receive buffer.
                colIdx = find(JremoteNZ==recv_data_matrix(recv_data_matrix_idx(i),i));
                recv_buff(1:colCnt(recv_data_matrix(recv_data_matrix_idx(i),i))*2,colIdx) = MPI.C
                recv_data_matrix_idx(i) = recv_data_matrix_idx(i) + 1;
                if recv_data_matrix_idx(i) <= size(recv_data_matrix,1)
                    if (recv_data_matrix(recv_data_matrix_idx(i),i) > -1)
                        % We need to ask processor i for another column. Send the request.
                        q = MPI.COMM_WORLD.isend([1 recv_data_matrix(recv_data_matrix_idx(i),i)],2,i-
                        MPI.COMM_WORLD.waitReq(q,false);
                        q = MPI.COMM_WORLD.irecv(colCnt(recv_data_matrix(recv_data_matrix_idx(i),i))*
                        i-1,dataRecvTag);
                        recv_data_wait_list(i) = q;
                    else
                        recv_data_wait_list(i) = -1;
                    end;
                else
                    recv_data_wait_list(i) = -1;
                end;
            end;
        end;
    end;
end;

```

```

        end;
    end;
end; % end of for that checks for data receives.
% Now, check to see if we have received any requests from any other processors.
for i = 1 : nproc,
    if (reqWaitList(i) ~= -1)
        if (MPI.COMM_WORLD.testReq(reqWaitList(i)))
            messThere = true;
            mess = MPI.COMM_WORLD.waitReq(reqWaitList(i),true);
            op = mess(1);
            mess = mess(2);
            if (op == -1)
                % Processor i has finished its computation
                reqWaitList(i) = -1;
            elseif (op == 1)
                % Processor i is sending us a request for a column.
                [I J S] = find(A(:,mess-colIdxB+1));
                send_buff(1:length(I)*2,i) = [I ; S];
                q = MPI.COMM_WORLD.isend(send_buff(:,i),length(I)*2,i-1,dataRecvTag);
                MPI.COMM_WORLD.waitReq(q,false);
                send_data_wait_list(i) = -1;
                q = MPI.COMM_WORLD.irecv(2,i-1,dataRequestTag);
                reqWaitList(i) = q;
            end;
        end;
    end;
end; % end of for that checks for data requests
end; % end of while

```

## 4.5 Analysis

We have used the above code to generate sparse matrices for a smattering of tiny, random, sparse matrices. The results were correct (i.e. they matched those of the sequential version) We also used it to find the approximate inverse of `orsirr2`, an  $886 \times 886$  matrix from the Harwell-Boeing collection. We chose `orsirr2` because it is the primary matrix used for analysis in papers [1], [3], [5], and [6].

We determined the results were correct, but that the time-space trade-off was not yet acceptable (30[s] for this version, under 2[s] for the sequential version). This is to be expected, because we have no optimizations in the our code. Also, this matrix is still relatively small, and we intend our code to benefit large matrices only.

Our testing and analysis of the SPAI code were curbed by an unstable infrastructure. There seems to be a problem with `mm` and its interaction with matrices. We originally thought the instability was caused by the new MPI code and by our Matlab code, but have found that we can render Matlab \*P inoperable by issuing repeated calls to `mm`, `matlab2pp`, and `pp2matlab` without ever initializing the Java MPI interface or calling SPAI.

However, our experience designing the communication code and our observation that it is slow leads us to believe that we should consider another parallelism model. Matlab \*P does not currently have support for a global address system (GAS), but it may in the near future. Once that has been developed, we can use a global read-only pointer to access data from  $A$ . The code will be simpler, but it will not be trivial. We must still include some form of caching and latency-hiding. Also, we can explore using the code from [7] for the adaptive mode, and write code for the fixed mode that does count on a communication pattern that can be known ahead of time.

## 5 Future Work

### 5.1 Fix MM

Clearly, this is the most pressing issue. The infrastructure needs to be stable.

### 5.2 Caching

We can reduce the communication, if we implement a cache to store remote columns. We must limit the size of the cache, because our overall goal is to keep the size requirements to a minimum. This will be beneficial whether we continue to use the message-passing model, or move to a global address space model.

### 5.3 Add Support for Adaptive Mode

For this implementation to be complete, it must include the code for the adaptive mode. This will be relatively straight-forward to write. The most complicated part will be to handle the communication to acquire all the data necessary to complete Step 4 of the algorithm outlined above:

4. Set  $\tilde{\mathcal{J}}$  equal to the set of all new column indices of  $A$  that appear in all  $\mathcal{L}$  rows but not in  $\mathcal{J}$ .

Since we are distributing  $A$  by columns, each processor will have part of each row in  $\mathcal{L}$ . Finding the columns with nonzero entries will require communication with every other processor. This means, we need to perform additional bookkeeping and possibly caching for rows.

### 5.4 Load Balancing

If we can develop a way to perform the algorithm without `mm`, or if `mm` is altered to allow varying sizes of return matrices, then we can add support for load balancing.

### 5.5 Use Sparse Matrices in all Matlab \*P Code

Right now, `mm`-mode cannot send or return sparse matrices. Returning a dense matrix is expensive, so we can switch to sparse matrices as soon as that feature is available.

## 5.6 Perform More Extensive Tests

We would like to compute the sparse approximate inverse of larger matrices on Matlab \*P. It would be interesting to profile the new communication code to see what aspects of it are taking the most time.

## References

- [1] Grote, M. and Huckle, T. (1997). Parallel Preconditioning with Sparse Approximate Inverses. *SIAM Journal of Scientific Computing Vol. 18, No. 3* Pp. 838-853.
- [2] Benzi, M. (2002). Preconditioning Techniques for Large Linear Systems: A Survey. *Journal of Computational Physics 182*. Pp. 418-477.
- [3] Taylor, S. (2004). Sparse Approximate Inverses in Matlab \*P. <http://www.cs.ucsb.edu/~staylor/SPAI/spaiPaper.pdf>.
- [4] Goldman, M., Guo D. (2003). Java MPI in MATLAB \*P. <http://beowulf.lcs.mit.edu/18.337/>
- [5] Barnard, S., Bernardo, L., Simon, H. (1997). An MPI Implementation of the SPAI Preconditioner on the T3E. *The International Journal of High Performance Computing Applications*.
- [6] Barnard, S., Grote M. (1999) A Block Version of the SPAI Preconditioner. *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*
- [7] SPAI 3.1. <http://www2.inf.ethz.ch/personal/broecker/spai/>
- [8] Matlab\*P: Interactive supercomputing. <http://www.cs.ucsb.edu/~gilbert/StarPjan03.htm>.