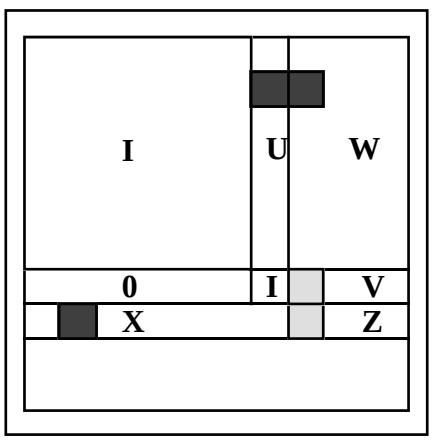
Technical Report CS-94-03

Department of Computer Systems
Faculty of Mathematics and Computer Science
University of Amsterdam

Implementation of Gauss-Huard's method on the IBM 9076 SP1

W. Hoffmann
G. G. Pronk

February 1994



Contents

The project is part of research on novel parallel numerical algorithms in the Parallel Scientific Computing Group of the Department of Computer Systems.

Abstract
In this paper, the implementation of a variant of Gaussian elimination, Gauss-Huard elimination, on the IBM 9076 SP1 is discussed. As programming environment, PVMe is chosen.

Keywords
Gaussian elimination, Parallel Virtual Machine, parallel algorithms, Gauss-Huard elimination.

Introduction

In 1979, Huard proposed a variant of Gaussian elimination, called "méthode des paramètres". This method reduces the given matrix to diagonal form and requires $2/3n^3 + O(n^2)$ floating point operations. Dekker proved that implementation of Gauss-Huard, with appropriate pivot strategy, is as stable as Gauss-Jordan elimination [Dekker93].
To reduce cache-operations in shared-memory systems, Hoffmann introduced a variant of the block Gauss-Huard algorithm: the twined block algorithm [Hoffmann93]. For this block algorithm, research on different parallel computers has already been done, see e.g. [Pronk93]. The emphasize of this project was to investigate how the twined block algorithm will perform on the IBM 9076 SP1. The expectation is that the architecture of the SP1 is efficient for this algorithm.

1. Gauss-Huard elimination

An advantage of the Gauss-Huard algorithm is that in the k^{th} step the upper left submatrix of order k is transformed into the identity matrix but the remaining n-k rows of the matrix are still unchanged. In the description of the block algorithms, A[i,j] refers to block[i,j] of matrix A instead of element[i,j]. For a more detailed description of the 'point' Gauss-Huard elimination see [Hoffmann93].

1.1 Description of the block algorithm

The block algorithm can be depicted as follows:
(in which 'n' is the dimension of the matrix and 'bs' the dimension of the block and 'nb'=n/bs is the total number of blocks in each direction).

```
1   for k = 1 to nb do
2       A[k, k:nb] = A[k, k:nb] - A[k, 1:k-1] x A[1:k-1, k:nb]
3       Apply Gauss-Huard with column-interchanges to A[k, k:nb]
4       Apply interchanges in upper and lower blocks
5       A[1:k-1, k+1:nb] = A[1:k-1, k+1:nb] -A[1:k-1, k] x A[k, k+1:nb]
6   enddo
```

fig. 1.1.1

The block algorithm is very similar to the point Gauss-Huard algorithm [Hoffmann93], [Pronk93]; except for elements, the algorithm is performed on blocks. The pivot steps however are different: on the diagonal-block (the 'pivot-block'), the point Gauss-Huard algorithm is performed to obtain an identity block. The accompanying column interchanges, due to pivoting, are performed on the whole matrix! Note that we can consider the right-hand side vector as an extra column of the matrix.

We will call step 2 of the block algorithm (fig. 1.1.1) also: "block row elimination" (Z:=Z–X*Y, see fig. 1.1.2) and step 5: "block column elimination" (W:=W-U*V, see fig. 1.1.3).

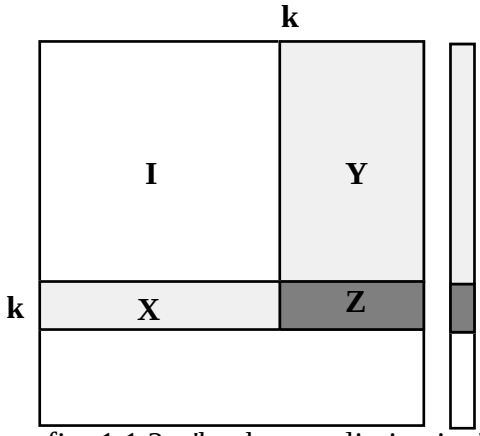


fig. 1.1.2 'bock-row elimination'

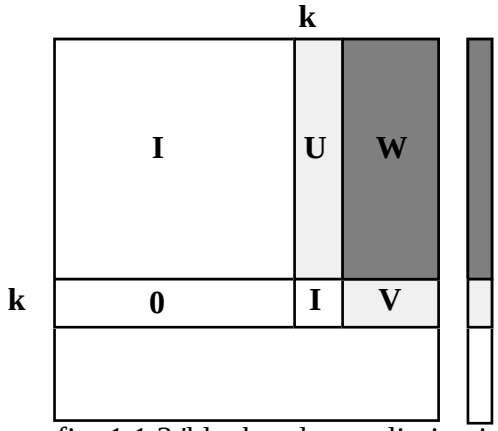


fig. 1.1.3 'block-column elimination'

Basic tasks for block-implementation on a shared memory machine
We can distinguish the following tasks for each loop step ($1 \le k \le nb$):

Block row elimination (step 2 of 1.1.1):

1. send first k-1 blocks of block-row k ($A_{[k,1:k-1]}$) to blocks in upper k-1 block rows;
2. load row blocks ($A_{[k,1:k-1]}$) and upper-matrix block ($A_{[1:k-1,k:nb]}$) in cache ;
3. perform block x block operation and update result block ($A_{[k,k:nb]}$);
4. summate over columns for update of row k ($A_{[k,k:nb]}$);
5. store result-block in memory;

Diagonal block (steps 3 and 4 of 1.1.1):

1. perform point Gauss-Huard algorithm (see 2.1.1) on diagonal and following blocks ($A_{[k,k:nb]}$);
2. perform column interchanges on upper and lower blocks;

Block column elimination (step 5 of 1.1.1):

1. broadcast column block ($A_{[1:k-1,k]}$) over blocks in upper k-1 rows;
2. broadcast row block ($A_{[k,k+1:nb]}$) over blocks in nb-k columns;
3. load blocks of block-update and result-block in cache;
4. perform block-update;
5. store result-block ($A_{[k,k+1:nb]}$) in memory;

The consequence of those steps is that the cache must contain minimal three blocks.

1.2 Twined block algorithm

The twined block variant is actually the same as the block algorithm discussed before. In this variant the block column elimination and the block row elimination are now chained. This may be efficient for implementation on shared memory machines with cache. If we combine the two block algorithm steps (i.e. block column elimination and block row elimination) we achieve a better cache usage; for this twined version two cache strategies are possible: columnwise (fig. 1.1.4 and fig. 1.1.5) and rowwise (fig. 1.1.6 and fig. 1.1.7) accessing of the blocks.

Twined elimination: columnwise accessing

($Z = Z - X*\frac{W - UV}{V}$; fig. 1.1.3)

step 1: In the first step, five blocks are loaded: from U,V,W,X and Z (fig. 1.1.6). Next, the block-update for W (W=W-U*V) is performed. W is overwritten with its own result. With the updated W-block, the Z-block can be updated. Also Z can be overwritten with its result. After those operations, the new W-block has to be stored. So five blocks are loaded and one block is stored.

step 2: The blocks of V and Z stay in cache; the following column blocks of U and W and the next block of X are loaded (fig. 1.1.6). Again, at first the block-update for W is performed and afterwards the update for Z is performed. Just as in the previous step, the new W-block is stored. For this step, three loads and one store are needed. For the last row, the updated Z-block has to be stored, so then one load (only the X-block) and one store (of Z) are needed.

This gives a total of $4jk - 4j^2$ load/store operations for one loop step. If we add up this result over j (all loop steps) we obtain the amount of load/store operations needed for the block-column and block-row updates of the whole algorithm: $2/3k^3$ load/store operations.

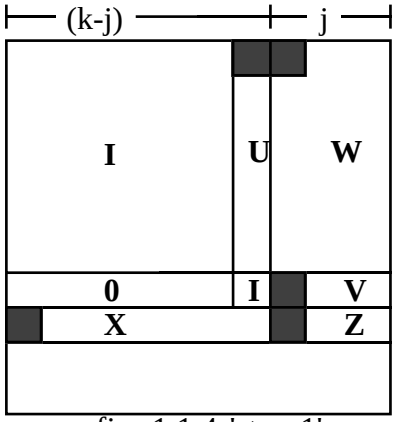


fig. 1.1.4 'step 1'

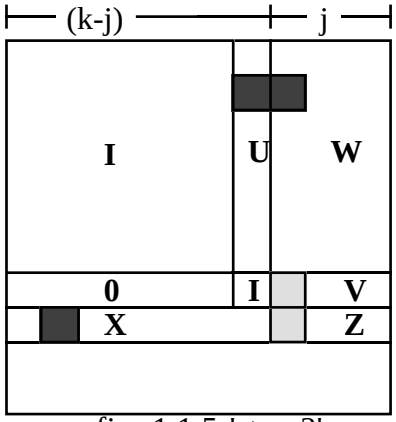


fig. 1.1.5 'step 2'

Twined elimination: rowwise accessing

($Z = Z - X*\frac{W - UV}{V}$; fig. 1.1.3)

step 1: In the first step, five blocks are loaded: from U,V,W,X and Z (fig. 1.1.6). Next, the block-update for W (W=W-U*V) is performed. W is overwritten with its own result. With the updated W-block, the Z-block can be updated. Also Z can be overwritten with its result. After those operations, the new W and Z-block have to be stored. So five blocks are loaded and two blocks are stored.

step 2: The blocks of U and X stay in cache; the following row blocks of V, W and Z are loaded (fig. 1.1.7). Again, at first the block-update for W is performed and afterwards the update for Z is performed. Just as in the previous step, the new blocks of W and Z are stored. For this step, three loads and two stores are needed. For the last row however, the W block is not updated, so two loads (the V and Z block) and one store (of Z) are needed.

This gives a total of $5jk+5k-5j^2-15j-6$ load/store operations for one loop step. If we add up this result over j (all loop steps) we obtain the amount of load/store operations needed for the block-column and block-row updates of the whole algorithm: $(5/6)k^3-(5/2)k^2+5k-6$ load/store operations.

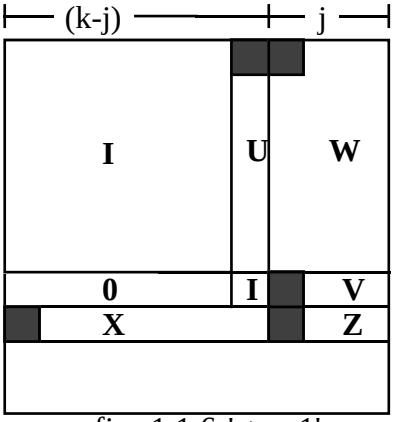


fig. 1.1.6 'step 1'

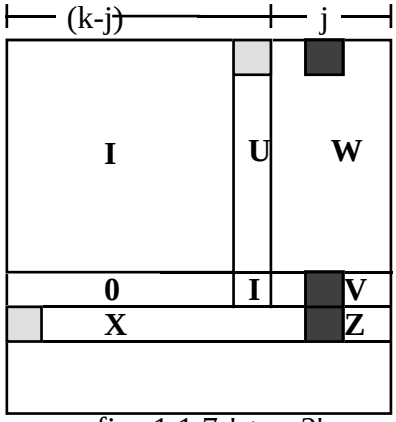


fig. 1.1.7 'step 2'

Conclusions:
The columnwise accessing of the twined elimination uses about $(1/6)k^3$ less load/store operations. Compared with the results of the non-twined elimination steps, the twining of the two steps can reduce the amount of load/store operations with roughly a quarter if the columnwise accessing of data is used.

2. IBM 9076 SP1

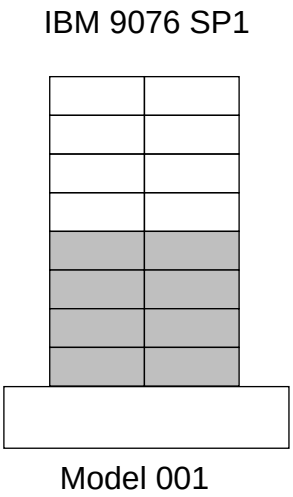


fig. 2.1 IBM 9076 SP1, model 001

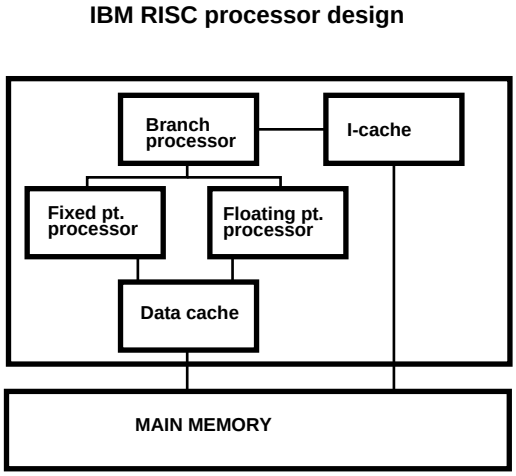


fig. 2.2 RISC 6000 processor

The IBM SP1 contains of 8 nodes (62.5 MHertz RISC/System 6000).The nodes are connected with a so-called "High Performance Switch (HPS)"; the HPS takes care of a fast point-to-point communication: the latency is 500 nanoseconds and the peak bandwidth is 40 MBytes. However the actual bandwidth is at most 8 MBytes [Vermin94]!

In figure 2.2, an IBM RISC 6000 processor is depicted. Each processor has one GByte disk and 64 MByte of main memory. The data- and instructioncache have both the size of 32 KByte. The floating-point processor contains of a two-stage pipeline: first a multiplication and next an addition can be performed in one cycle.

3. PVMe

PVM is a public domain package that is developed at the Oak Ridge National Laboratory. To make use of the High Performance Switch, IBM developped an enhanced PVM version: PVMe(nhanced). This version is based on the PVM 2.4.2 version. A main difference between PVM and PVMe is that the first one can be considered as a statefull system while PVMe is a stateless system.
For more information about the differences between PVM and PVMe, see [IBM93].

4. Implementation

Data distribution

We can distinguish four different data distributions (there are more possibilities but we restrict ourselves to these four):
1. forward diagonal distribution (fig. 4.1)
2. backward diagonal distribution (fig. 4.2)
3. horizontal distribution (fig. 4.3)
4. vertical distribution (fig. 4.4)
(In the figures, the distribution of data over three processors (three colours) is depicted.)

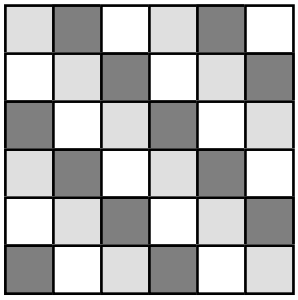


fig. 4.1 forw.diag. distrib.

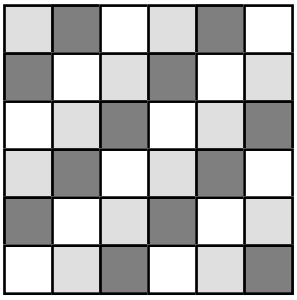


fig. 4.2 backw. diag. distrib.

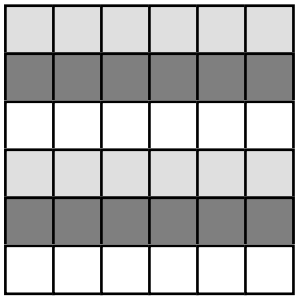


fig. 4.3 hor. distrib.

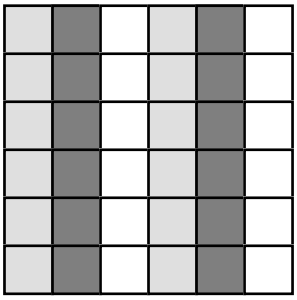


fig. 4.4 vert. distrib.

There are four kind of communications that we use (with increasing complexity):
a. send from one processor to another processor: single send
b. broadcast from one processor to all other processors: single broadcast
c. all processors send to one processor: multiple send
d. all processors broadcast to all processors: multiple broadcast
Now we can make a theoretical assumption about the communication that is needed for the four data distributions.

Block elimination
pivoting needed for calculation of the identity-block:
1,2 and 4: communication needed in each block row to determine the max. element; single and multiple sends are needed
3: no communication needed because all blocks are in the same processor

update lower and upper blocks in same column:
1,2 and 3: single broadcast needed to distribute swap-indices
4: no communication needed because all blocks are in the same processor

Block row elimination
blocks of Y to processors that contain blocks of W and V:
1,2 and 4: multiple broadcast
3: single broadcast (all blocks of Y in one processor)

blocks of W and V to processors that contain blocks of Z:
1,2 and 3: multiple broadcast
4: no communication needed (all blocks are in one processor)

Block column elimination
blocks of U to processors that contain blocks of W:
1 and 2: multiple broadcast
3: no communication needed
4: single broadcast

blocks of V to processors that contain blocks of W:
1 and 2: multiple broadcast
3: single broadcast
4: no communication needed

Conclusion
The two diagonal distributions are equally bad. We have chosen the horizontal distribution because of less communications during pivoting. For more information about the theoretical performance analysis, see [Hoffmann93] and [Pronk93].

5. Difficulties

Unfortunate, the PVMe version on the IBM SP1 did not work. So no results with the High Performance Switch are obtained. Because the performance of PVM 2.4.2 was also not smooth and because lack of time, the test program could not be tested. We propose to postpone the execution of the program until the PVM(e) programming environment is implemented satisfactorily.

Conclusion

Until the end of the current project, February 15, 1994, PVM(e) appeared not to be operational on the IBM SP1 installed at SARA. We preferred not to consider the use of other software for parallelization on the IBM SP1.

Acknowledgements

We want to thank miss Drs. K. Potma for her contribution to the program code. We also want to thank Willem Vermin from SARA, for supporting our effort to get our code running on the IBM 9076 SP1 computer.

References

[Dekker89] Dekker, T.J. & Hoffmann, W.; "Rehabilitation of the Gauss-Jordan algorithm, *Numerische Mathematik* **54**, 1989, p591-599.

[Dekker93] Dekker, T.J., Hoffmann, W. and Potma, K.; "*Stability of Gauss-Huard Elimination for Solving Linear Systems*", Technical Report CS-93-08, 1993, Faculty of Mathematics and Computer Science, University of Amsterdam.

[Hoffmann89] Hoffmann, W.; "*Basic Transformations in Linear Algebra for Vector Computing*", 1989, Thesis, University of Amsterdam.

[Hoffmann93] Hoffmann, W., Potma, K. and Pronk, G.G.: "Solving dense linear systems by Gauss-Huard's method on a distributed memory system", to be published in Future Generation Computers, Amsterdam.

[Huard79] Huard, P; "La méthode simplex sans inverse explicite", *Bulletin de la D.E.R. - E.D.F.*, **Série C**, **n°2**, 1979, p79-98.

[IBM90a] Language Reference for IBM AIX XL FORTRAN Compiler/6000 Version 2.3, third ed., 1990.

[IBM90b] User's guide for IBM AIX XL FORTRAN Compiler/6000 Version 2.3, third ed., 1990.

[IBM92] Optimization and Tuning Guide for the IBM XL FORTRAN and XL C Compilers, first ed., 1992.

[IBM93] IBM AIX PVMe User's Guide and Subroutine Reference, 1993.

[Potma89] Potma, K.; "*Huard's Method for Solving Linear Systems on Vector and Parallel Vector Computers*", Technical Report CS-89-12, 1989, Faculty of Mathematics and Computer Science, University of Amsterdam.

[Potma93] Potma, K.; "*Numerical Experiments to Support the Error Analysis of the Gauss-Huard Algorithm with Partial Pivoting*", Technical Report CS-93-09, 1993, Faculty of Mathematics and Computer Science, University of Amsterdam.

[Pronk93] Pronk, G.G.; "*Performance analysis and implementation of the Gauss-Huard algorithm*", Master's thesis, 1993, University of Amsterdam.

[Vermin94] Vermin, W.; unpublished information about the IBM 9076 SP1, SARA, Amsterdam.

Appendix: source codes

The program of Gauss-Huard elimination considers a host and a node program; resp. called "myhost" and "mynode". The code is written in PVMe and Fortran77 and makes use of the BLAS library. Parts of the code, the calculation of an identity block ("ghvu") and some supporting routines ("gentst", "ranmat" and "ranvec"), were already written by miss Drs. K. Potma.