Solving Large Dense Systems of Linear Equations on Systems with Virtual Memory and with Cache

Walter M. Lioen and Dik T. Winter
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The problem of solving large full sets of linear equations on a computer with hierarchical memory is considered. Blocked Gaussian elimination and QR factorization are studied in an attempt to apply exactly the same implementations on computers with virtual memory as on computers with cache. This differs slightly from the LAPACK[1] approach which, although aimed at the same architectures, focuses mainly on computers with vector registers and/or cache. Experiments on a Cyber 205 and an Alliant FX/4 are reported.

1980 Mathematics subject classification (1985 revision): 65V05, 65F05, 65F25, 68M20.
1987 CR Categories: C.1.2, G.1.3.

Keywords & Phrases: Linear equations, block algorithms, cache, virtual memory.

1. Introduction

Since the introduction of digital computers there has always been the need for more fast memory than physically available. Back in 1953 Wilkinson [16] had already described the solution of linear systems larger than order 13 (which was the limit for the central memory on the Pilot ACE) using punch cards as secondary storage. In [2] magnetic tapes are used as secondary storage. At present Grimes [10] extrapolates for linear systems up to order 80,000 using SSD or disks on a Cray X-MP and a Cray 2, respectively.

In this paper we will discuss the overall performance (both with respect to the CPU time and the IO time, resulting in the wall-clock time) of blocked Gaussian elimination and QR factorization on a virtual memory system. For our measurements we have used a locally available Cyber 205 because of its easily predictable and measurable page replacement algorithm, but results will hold both for any virtual memory system with a suitable page replacement algorithm and for implementations using explicit IO.

Finally, we will discuss the behaviour of the presented blocked Gaussian elimination algorithm on a computer with cache. This differs slightly from the LAPACK[1] approach which, although aimed at the same architectures, focuses mainly on computers with vector registers and/or cache. For this experiment we used an Alliant FX/4.

2. Virtual Memory

Before we are going to study the behaviour of algorithms in a virtual memory environment, we have to explain what virtual memory is, and, how it behaves.

Fast memory always has been and always will be an expensive part of a computer system. One of the solutions for this problem is to pretend there is much more memory than physically available by using (much) slower disk(s). Programs from now on have to use virtual addresses instead of the physical addresses. The hardware and/or the operating system translate a virtual address to a physical address and check whether the requested datum is already available in physical memory. Otherwise a so-called *page-fault* occurs: the hardware has to fetch the datum from disk. Because of the disk-seek latency this transport is done in so-called *page*s corresponding with one or more disk blocks.

Report NM-R9123
ISSN 0169-0388
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

This process is repeated until all physical memory is in use and if the latter is the case, we have to make use of a so-called *page replacement algorithm*:

```
select a page, e.g., the **L**east **R**ecently **U**sed;
if the data on this page has been altered
  then we have to write the data back to disk
  else we can discard this page without writing;
from this moment on we can safely re-use the selected space.
```

2.1. Algorithms and virtual memory

What is the impact of virtual memory on the behaviour of algorithms? Suppose we have a computer with a page-size of 100 words and a memory consisting of 1000 words (i.e. 10 pages). Furthermore we assume a Least Recently Used page-replacement algorithm.

We are going to study an algorithm which is running linearly through a problem with a dataset size of 1024 words. The data itself occupies 11 pages; this is 1 page too much. One application of this algorithm gives rise to 11 page faults.

However, applying the same algorithm N times gives rise to $11N$ page faults:

1	2	3	4	5	6	7	8	9	10	11

If we are running the first time through our data we have to page in the page numbered 1 (which we from now on will address to as page 1) up to and including page 10. At the moment we have to page in page 11, we have to discard a previous page. Since we assumed LRU, we discard page 1 (and if the data on page 1 has been altered we have to write page 1 back to disk); now page 11 can be paged in.

All following times we are running through the same data we have to page in the page we have just discarded in the previous step (e.g., at the moment we have to page in page 1, we have just discarded this page because we had to page in page 11; at the moment we have to page in page 2, we have just discarded this page because we had to page in page 1; and so on). This phenomenon is often referred to as *page thrashing*.

Of course, a solution for this paging problem may be found by reorganizing the algorithm.

3. Large dense systems of linear equations

First of all we have to define the adjective large: with large we mean a factor larger than physical memory. For current midrange computers say of the order $n = 2000,\ 4000,\ 8000$.

Our main concerns are: numerical stability and wall-clock time. Numerical stability is very important because for large n it can in general not be known a priori whether the final result will have any significance! Wall-clock time is another important issue because the CPU-time is of order n^3 and the IO-time is also of order n^3.

Because of the IO involved in iterative methods (k steps of an iterative method will in general use order kn^2 IO-time) we only consider the following direct methods (cf. [9]).

3.1. Gaussian elimination

The computational work involved in an L(D)U factorization is of order $\frac{2}{3}n^3 + O(n^2)$. Complete pivoting is practically impossible due to the IO involved. Partial pivoting is possible, but for large problems it is unknown whether Gaussian elimination is stable enough! It is possible to start the Gaussian elimination with partial pivoting and monitor the growth factor [3]. If the monitored growth factor becomes too large, one can switch to complete pivoting, but as mentioned above this will be practically impossible due to the IO involved. Iterative improvement also has some practical difficulties: no improvement may result if the original matrix is too ill-conditioned; moreover, one also needs the original matrix for the residual computations, which will roughly double the IO-time for one iterative improvement step.

3.2. QR factorization

Rank Revealing QR factorization [4] is not possible because of the IO induced by the pivot search (column interchanging). For the orthogonalization we have four possibilities:

(Modified) Gram-Schmidt $2n^3 + O(n^2)$

Householder $\frac{4}{3}n^3 + O(n^2)$

Givens $\frac{8}{3}n^3 + O(n^2)$

Fast Givens $\frac{4}{3}n^3 + O(n^2)$

In section 6 we will explain why we use Householder reflections.

3.3. Singular Value Decomposition

For near singular systems this is theoretically the most sound method. However, the computational work (for a full Golub-Reinsch SVD) is of order $(14 + \frac{22}{3})n^3 + O(n^2)$ which is too much because n is large. Also, the amount of IO needed becomes unacceptably large.

3.4. Which method to choose?

The advantage of QR factorization over Gaussian elimination is its intrinsic numerical stability because of the use of orthogonal transformations. However, the computational work is twice as much. In the following sections we show that both methods can be reorganized using an identical IO structure. As we will show in Section 5, complete monitoring of the growth factor [3] will be possible for blocked Gaussian elimination only by introducing extra IO. Since we want to minimize the IO, the growth factor can be computed only after the last column has been updated. In our algorithm, however, we monitor only the growth factor of the columns being updated. Thus, an early abort of Gaussian elimination will sometimes, but not always be possible (worst case: growth in last column). Ultimately, we can even abort the Gaussian elimination process and restart with a QR factorization. The eventual choice will depend on the problem at hand.

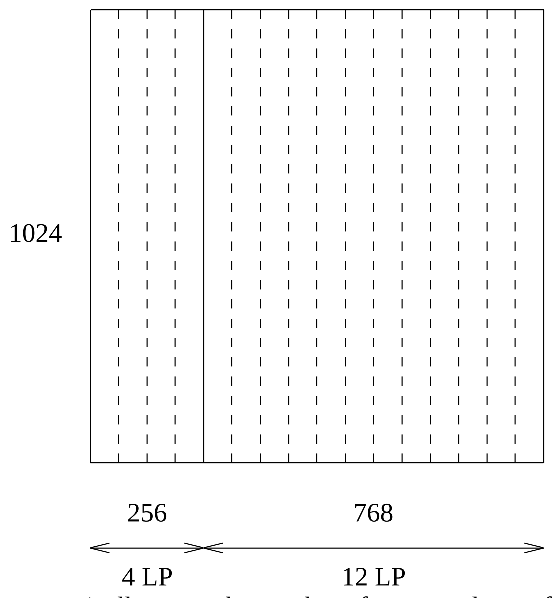
4. NON-BLOCKED GAUSSIAN ELIMINATION

Before we are going to discuss blocked Gaussian elimination we first describe an experiment we performed with a non-blocked Gaussian elimination implementation. The machine we used was a 1-pipe Cyber 205. The (Large) Page size of the Cyber 205 is 65536 words and the Cyber 205 uses an LRU page-replacement algorithm. The Central Memory at the time was 1 Mword, and the associated maximal Working Set was 12 (Large) Pages.

We wanted to solve a 1000×1000 system (which occupies more than the available maximal Working Set) with the fastest *in core* algorithm at our disposal: CCRPCF (see [11]). This is an LDU-decomposition with diagonal scaling $l_{ii} = u_{ii} = d_i^{-1}$, and with column interchanging.

```
for $k=1, \cdots, n$ do
   perform pivotal search (use maximal $A_{kk}$)
   interchange columns/rows
   $A := A - A_{kk}^{-1} A_{[k+1:n,k]} \times A_{[k,k+1:n]}$
   (this is a rank-1 update, a BLAS 2[6] routine)
end do
```

4.1. The solution of a 1024×1024 *system, analysis*
For a 1024×1024 system exactly 64 columns fit on 1 Large Page and the total problem needs 16 Large Pages. It is relatively easy to predict the number of page faults. After 256 *k*-steps of the algorithm (the columns in the first 4 Large Pages) all work can be done in core (the last 12 Large Pages).



In the following table we systematically count the number of generated page faults:

page faults per columns	pivotal search	rank-1 update
1st 64 columns	16	16
2nd 64 columns	15	15
3rd 64 columns	14	14
4th 64 columns	13	13
all following columns	in core	in core

This totals up to $64 \times (2 \times (16+15+14+13)) = 7424$ LP faults.
For simplicity we neglect the actual (problem dependent) column interchanging.

4.2. The solution of a 1000×1000 *system, actual experiment*
The following timings were obtained on a 1-pipe Cyber 205.

CPU time	8.9 sec	
IO time	$\pm$ 1 hour	(7431 LP faults $\times 0.5 \frac{\text{sec}}{\text{LP fault}}$)
wall-clock time	6 hours	(61 times suspended)

However, by applying the block algorithm we will present in the following section, the same problem could be solved in approximately:

CPU time	8.9 sec	
IO time	$\pm$ 10 sec	(20 LP faults)
wall-clock time	20 sec	

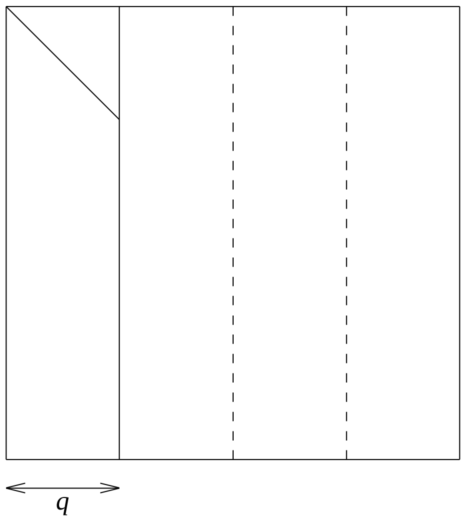
So, ‘blocking’ the Gaussian elimination roughly reduces the wall-clock time by a factor of 1000.

5. Blocked Gaussian Elimination

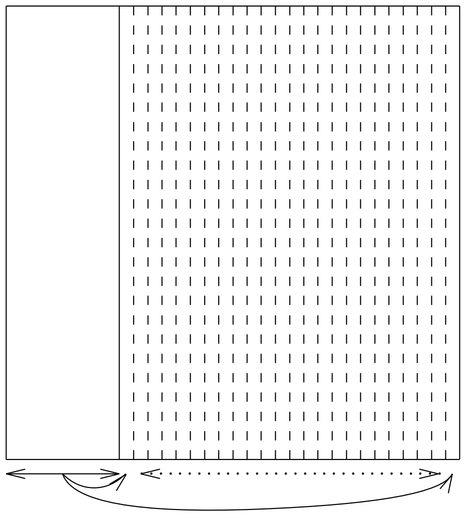
In the following we assume an LRU page-replacement algorithm, otherwise we have to make use of explicit IO (reading and writing of files or using machine specific asynchronous IO).

In order to reduce the IO we partition the matrix in vertical strips of q columns, where $q+1$ columns will fit in main memory. However, the most important algorithmic choice, in the LDU decomposition itself, is the use of row interchanging; otherwise, a sweep through the complete matrix has to be made for every pivotal search step.

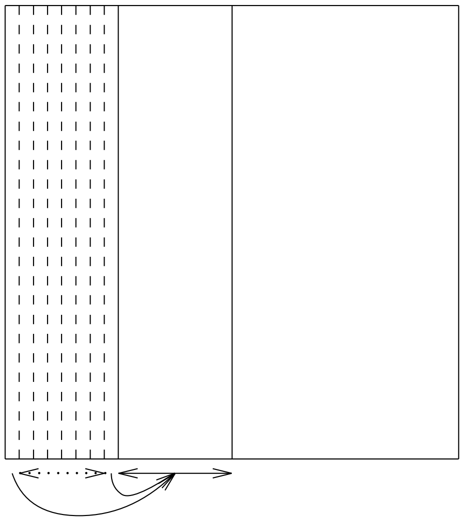
Step 1: Restrict all work to be done for the LDU decomposition to the first q columns, and postpone all row interchanging and updates outside those q columns. With our previous choice of q all this work (step 1) can be done in core. If q is not a proper divisor of n then we take the first $n \bmod q$ columns instead of q columns.



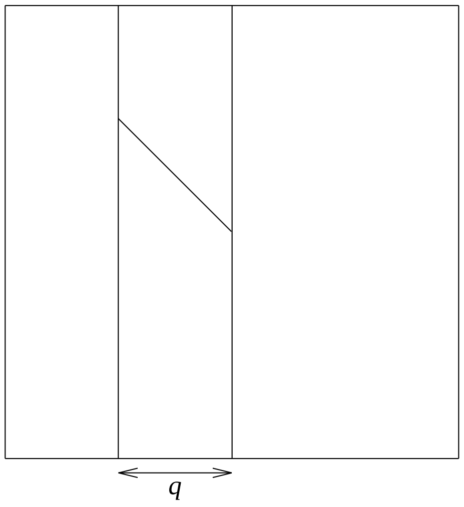
Step 2 (first possibility): Apply the previous interchanges and updates successively to the columns $q+1, \cdots, n$. This variant is known as right-looking. During this step the first q columns stay in core and only the to-be updated column varies. The disadvantage of this approach is that the successive columns have to be updated: they have to be read as well as written.



Step 2 (second and superior possibility): Successively apply the previous interchanges and updates to the 2nd q columns $q+1, \cdots, 2\times q$. This variant is known as left-looking. Here the second q columns will stay in core and only the columns from the previous strip $1, \cdots, q$ vary. The advantage over the first possibility is that the previous columns only have to be read. The disadvantage for blocked Gaussian elimination is, that complete monitoring of the growth factor [3] is no longer possible (It was in the previous possibility, but at the cost of extra IO). In this case it is only possible to monitor the growth factor of the columns being updated, so an early abort of Gaussian elimination will sometimes, but not always be possible (worst case: growth in last column).



Step 3: Restrict all work to be done for the LDU decomposition to the 2nd q columns $q+1, \cdots, 2\times q$. Again we postpone all row interchanges and updates outside those q columns.



Finally, repeat steps 2 (second possibility) and 3 on the appropriate columns until the whole matrix has been treated.

The IO of the original algorithm is of order n^3. It can easily be seen that the IO is reduced by a factor q (cf. Du Croz et al. [7]) as opposed to the original algorithm, but it is still of order n^3!

5.1. Blocked Gaussian elimination, analysis

In [7, 14, 15] one can find an analysis of blocked Gaussian elimination. To keep this kind of analysis simple, one has to make some simplifications.

The following assumptions yield exact results for the prediction of the number of page faults in the experiments described in Section 7. In the following analysis only square $(n \times n)$ matrices and page sizes p (in words) satisfying $n \le p < n^2$ are considered (as in [14]). Furthermore, we assume the original matrix aligned on a page boundary and p a multiple of n. (So the *packed row storage* and *row storage* of [14] coincide.) The last two assumptions are stronger than in [7] where one starts making ‘crude assumptions’ and constructs approximate formulas which are asymptotically valid for large n.

Suppose q not a proper divisor of n:
for step 1 ($k=0$) we have to make $(n \bmod q)\dfrac{n}{p}$ page faults; and
for $k=1, \cdots, \lfloor n/q \rfloor$ (steps 2 and 3) we have to make $q\dfrac{n}{p} + (n \bmod q)\dfrac{n}{p} + (k-1)q\dfrac{n}{p}$ page faults.
Summation yields the following total for the number of page faults:

$$\left(1+\lfloor n/q \rfloor\right)(n \bmod q)\frac{n}{p} + \sum_{k=1}^{\lfloor n/q \rfloor} kq\frac{n}{p} =$$

$$\left\{\frac{q\lfloor n/q \rfloor n}{2p} + (n \bmod q)\frac{n}{p}\right\}\left(1+\lfloor n/q \rfloor\right). \tag{5.1}$$

For q a proper divisor of n the analysis is somewhat different, but the result above still holds.
So, this formula then simply reduces to

$$\frac{n^2}{2p}\left(1+\frac{n}{q}\right),$$

which in turn approximately equals

$$\frac{n^3}{2pq}.$$

This is the same as the asymptotic result for $p \ge n$ [7, eqn. (3.1)]. A careful analysis yields that it is slightly favorable for the $n \bmod q \ne 0$ case to do $n \bmod q$ columns in step 1 as opposed to doing $n \bmod q$ columns in the last steps 2 and 3. This is a refinement of the algorithm described in [7].

6. BLOCKED QR FACTORIZATION

Every non-singular matrix A can uniquely be factorized as

$$A = QR,$$

where Q is an orthogonal matrix and R an upper triangular matrix.

During the QR factorization process the columns of A are orthogonalized with respect to all previous columns. Pivoting is not needed because the matrix A is assumed to be non-singular.

The IO structure of blocked QR is identical to that of Gaussian elimination: the first step we orthogonalize all columns in one strip; during the second step we orthogonalize - with respect to all previous columns - all columns in the next strip; in the third step we orthogonalize all columns in this next strip with respect to each other. Finally, as was the case for blocked Gaussian elimination, the last two steps are repeated until the whole matrix is done.
The orthogonalization process can be accomplished using

projections: Gram-Schmidt $2n^3 + O(n^2)$.

There are mainly two possibilities for performing the Gram-Schmidt algorithm [9]: CGS (Classical Gram-

Schmidt) and MGS (Modified Gram-Schmidt). Because of numerical stability MGS is preferred. However, even with MGS we eventually will loose orthogonality. The solution is iterating the MGS process: IMGS. Experiments of Hoffmann[12] indicate that 2 iterations suffice. This doubles both the CPU-time as well as the IO-time. Because our goal is minimizing the amount of IO, we have to use one of the following (numerically stable) orthogonal transformations:

reflections: Householder $\frac{4}{3}n^3 + O(n^2)$,
rotations: Givens $\frac{8}{3}n^3 + O(n^2)$,
rotations: Fast Givens $\frac{4}{3}n^3 + O(n^2)$.

Since the IO for QR using reflections and rotations is identical, we do not use Givens rotations because they involve twice as much computational work. Because of the possibility of element growth you have to monitor the matrix elements during Fast Givens to avoid overflow. The nontrivial overhead results in a QR algorithm that is slower than the Householder approach, so we will use Householder reflections.

7. Virtual memory experiments

As we already mentioned in our introduction we used a locally available Cyber 205 because of its easily predictable and measurable page replacement algorithm, but the results will hold both for any virtual memory system with a suitable page replacement algorithm and for implementations using explicit IO. For the following experiments we used a 2-pipe Cyber 205 with a central memory of 4 Mword and a maximal Working Set of 50 (Large) Pages. (As we mentioned before, the (Large) Page size p of the Cyber 205 is 65536 words and an LRU page-replacement algorithm is used). The algorithms we used are the above described blocked LDU and blocked QR. For our strip we use 47 of the 50 Large Pages. In general we need 2 pages for the columns we use for our delayed updates because such a column can cross a page boundary; furthermore, we need 1 page as workspace.

First we give the expected number of page faults using formula (5.1) and adding the matrix size in pages (the trivial number of page faults for the matrix initialization). For $n = 2048$ the matrix occupies 64 LP; exactly 32 columns fit on a page, our strip is 47 pages, so $k = 1504$; the computed number of page faults is 145. For $n = 4096$ the matrix occupies 256 LP; exactly 16 columns fit on a page, our strip is 47 pages, so $k = 752$; the computed number of page faults is 1087.

Experiment with blocked LDU

n	LP faults	IO time	CPU time	wall-clock
2048	177	89 sec	60 sec	184 sec
4096	1214	607 sec	356 sec	749 sec

Experiment with blocked QR

n	LP faults	IO time	CPU time	wall-clock
2048	145	73 sec	137 sec	200 sec
4096	1111	556 sec	891 sec	1341 sec

As we might expect from the computational complexity, the CPU time for QR is roughly twice as big as the CPU time for LDU. As predicted in section 6, the IO times are almost identical. If we compare the results of the blocked LDU with the results of the non-blocked LDU in section 4, we see that IO and CPU are now in balance.

To illustrate the sensitivity of the algorithms for the parameter q we present an accidental mistake as an additional experiment: the first time we ran the blocked LDU code we forgot that the workspace also took one Large Page. This resulted for n=2048 in 800 LP faults (instead of 177)!

8. Cache

Normally processor(s) are faster than the central memory. E.g., loading operands and storing results takes more time than performing a multiply. One of the solutions is using a cache, a piece of extremely fast memory between the processor(s) and central memory.

As can be seen from the following translation table cache behaves like virtual memory:

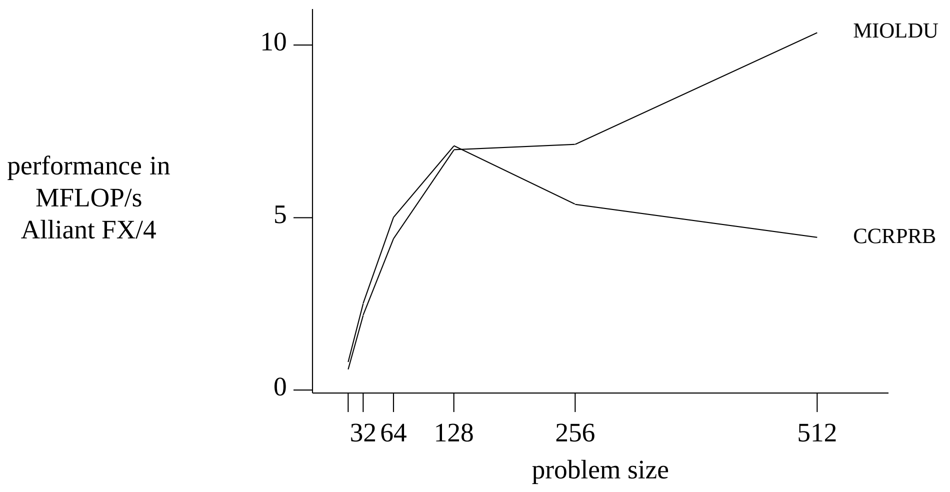
virtual memory	**cache**
physical memory	cache
disks	physical memory
page fault	cache miss
page	cache line
page replacement algorithm	cache policy

Since cache behaves like virtual memory we can apply exactly the same blocked algorithms we described before on a computer with cache.

9. Cache experiments

For the following experiments we used a locally available Alliant FX/4. We only used the Alliant here as an example for demonstrating the cache-effects.

Problems we encountered on the Alliant are: the algorithm has to be fast enough to see cache-effects and the cache-policy is not LRU: it behaves random. For a better description of the Alliant's cache see [17]. All results we present are based on 64-bit precision. The Alliant FX/4 cache size is 256k = 32 kwords, so matrices up to order 181 will fit in cache.



CCRPRB: non-blocked LDU decomposition
MIOLDU: blocked LDU decomposition

The performance drop we observe for CCRPRB is typical for a problem larger than the cache size of a given machine. The gradual decrease in performance indicates a random cache policy.

The maximal performance we obtain for MIOLDU, 10.4 MFLOP/s, is satisfactory in the sense that it is almost the peak performance of the Alliant FX/4 for the arithmetical operations used in our algorithm.

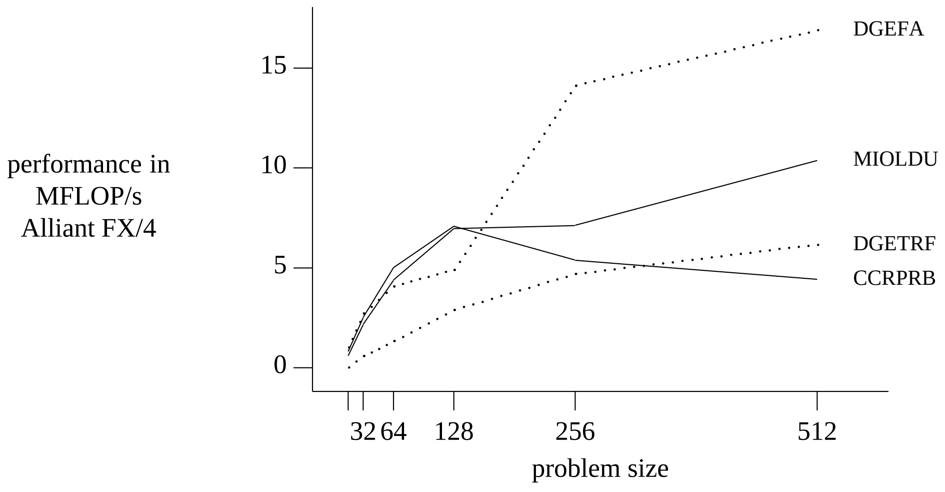
The most important observation, however, is that the same blocked LDU we used for virtual memory also gives satisfactory results for cache.

10. OTHER BLOCKING STRATEGIES.

One might ask whether vertical strips is an optimal blocking strategy. It is also possible to partition the matrix into square blocks.

This has the advantage that we can make use of matrix-matrix operations on the blocks instead of using rank-1 updates on strips (this is the BLAS 3 [5] approach); intermediate results can be kept in vector registers, so we can save result stores and operand loads.

In the following picture we compare our previous results with two fully blocked Gaussian elimination implementations on the Alliant FX/4



DGETRF:	preliminary result of (64)blocked LAPACK[1]
DGEFA:	an Alliant FX/4 tuned, blocked FX/Linpack result (Kuck et al)[13]
CCRPRB:	non-blocked LDU decomposition
MIOLDU:	blocked LDU decomposition

As expected DGEFA performs better than our blocked LDU implementation. The performance gain is already explained above and is not due to a decrease in IO, so, in this sense, it falls out of the scope of this paper.

11. CONCLUSIONS

In most computers there are more memory hierarchies than one in first instance might think of:

punch cards
paper tape
drum

magnetic tape
disk (virtual memory)
electronic disks: RAM disk / SSD / XMU / ECS
shared memory
local memory
cache
(vector-)registers

In this paper we presented blocked algorithms for a 2-level memory hierarchy. For 3-level memory hierarchies even nested blocking is analyzed in [8].

One might be tempted to say 'I have a Cray 2-512 (512 Mword) and can solve systems up to order 23170 in core! Why should we bother at all?' As we already showed in the introduction, history learns memory is always too small. Even more important is the fact that most people don't have access to a machine like a Cray 2-512.

More general: block partitioning allows for parallel computations on different blocks. Even on the above mentioned Cray 2 we greatly benefit from block partitioning (think of the performance of the FX/Linpack routine on the Alliant FX/4).

For local memory multiprocessors we have to minimize the slow inter-processor data-transport e.g., we can do domain-decomposition on a hypercube.

As we have shown with our experiments, one can use the same blocking-strategy for virtual memory systems as for cache memory systems. The opposite approach, using the same for cache memory systems well-behaving blocking strategy as for virtual memory systems, does not necessarily work! Our experiments differ from the LAPACK approach which, although aimed at the same architectures, focuses mainly on computers with vector registers and/or cache. The LAPACK project uses implicit blocking of the data and uses BLAS3 routines on the blocks. This works very nicely for cache memory systems; however, if one would use exactly the same algorithm on a virtual memory system one might again expect thrashing. Because of the relatively small cache line size and the relatively small cache miss penalty compared to the page size and the high page fault penalty, there is no direct need for explicit blocking of the original data for a cache memory system. It might be possible to construct implicitly blocked algorithms that behave well for virtual memory systems, by carefully accessing the matrix blocks; however, best performance is to be expected by explicitly blocking the matrix and storing the individual matrix-block-elements in contiguous memory.

Finally, in Section 5.1 we mentioned a small refinement of the algorithm described in [7].

References

1. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen (1990). *LAPACK Working Note #20: LAPACK: A Portable Linear Algebra Library for High-Performance Computers*, CS-90-105, University of Tennessee.
2. D.W. Barron and H.P.F. Swinnerton-Dyer (1960). Solution of Simultaneous Linear Equations using a Magnetic-Tape Store, *The Computer Journal*, 3.1, 28−33.
3. P.A. Businger (1971). Monitoring the numerical stability of Gaussian elimination, *Numerische Mathematik*, 16, 360−361.
4. Tony F. Chan (1987). Rank Revealing *QR* Factorizations, *Linear Algebra and its Applications*, 88/89, 67−82.
5. J.J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff (1990). A Set of Level 3 Basic Linear Algebra Subprograms, *ACM Transactions on Mathematical Software*, 16.1, 1−17.
6. J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson (1988). An Extended Set of FORTRAN Basic Linear Algebra Subprograms, *ACM Transactions on Mathematical Software*, 14.1, 1−17.
7. J.J. Du Croz, S.M. Nugent, J.K. Reid, and D.B. Taylor (1981). Solving Large Full Sets of Linear Equations in a Paged Virtual Store, *TOMS*, 7.4, 527−536.
8. Kyle Gallivan, William Jalby, Ulrike Meier, and Ahmed H. Sameh (1988). Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design, *The International Journal of Supercomputer Applications*, 2.1, 12−48.
9. G.H. Golub and C.F. Van Loan (1983). *Matrix Computations*, North Oxford Academic, Oxford.
10. Roger G. Grimes (1988). Solving Systems of Large Dense Linear Equations, *The Journal of Supercomputing*, 1, 291−299.
11. W. Hoffmann (1987). Solving linear systems on a vector computer, *Journal of Computational and Applied Mathematics*, 18.3, 353−367.
12. W. Hoffmann (1989). Iterative Algorithms for Gram-Schmidt Orthogonalization, *Computing*, 41, 335−348.

13. KUCK & ASSOCIATES, INC. (1988). *Para-Linpack/FX Users' Guide*, Release 2.1, Document #8809012
14. A.C. MCKELLAR and E.G. COFFMAN, JR. (1969). Organizing Matrices and Matrix Operations for Paged Memory Systems, *CACM*, 12.3, 153−165.
15. KISHOR SHRIDHARBHAI TRIVEDI (1974). *Prepaging and applications to structured array problems,* UIUCDCS-R-74-662, Phd. Thesis, University of Illinois at Urbana-Champaign.
16. J.H. WILKINSON (1954). Linear Algebra on the Pilot Ace, in *Automatic Digital Computation. Procs. of a symposium held at the National Physical Laboratory, March 1953*, Teddington.
17. DIK T. WINTER (1992, to appear). Influence of memory systems on vector processor performance, *Applied Numerical Mathematics*, 9.

-- --