STAMP

ASFRA
Voorhaven 33
1135 BL Edam
The Netherlands

SUPERCOMPUTER 55
volume X – number **3**
ISSN 0168 – 7875

volume **X** – number **3**
May **1993**

SUPER COMPUTER

55

Bimonthly magazine on supercomputing

SUPERCOMPUTER

SUPERCOMPUTER is the oldest journal devoted entirely to supercomputing. The first issue appeared in May 1984. The bimonthly journal primarily covers news and applications in the supercomputing field. It not only focusses on the largest machines, but also on other number crunching machines: minisupers, integrated and attached processors.

Articles are kept short, in general not exceeding 5–10 printed pages and contain material made readable for everyone interested in the field of supercomputers. The editorial team strives to keep the turn-around time for a contribution within a few months.

Instructions to Authors can be obtained from the editorial team. Articles can be sent to one of the editors. Contributions are refereed by at least two reviewers.

SUPERCOMPUTER is covered by ISI's Current Contents/Engineering, Technology & Applied Sciences and CompuMath Citation Index and Inspec.

Subscriptions: see backcover
© **ASFRA**, Voorhaven 33, 1135 BL Edam, The Netherlands.

Editorial Address:
SUPERCOMPUTER
P.O. Box 4613, 1009 AP Amsterdam, The Netherlands,
Telephone: +31 20 59 23 022; Fax: +31 20 66 83 167,
Electronic mail: SONDSCM@HASARA11.BITNET.

Editors:
Bill Buzbee *NCAR*
P.O. Box 3000, Boulder, CO 80307 USA
Jack Dongarra *University of Tennessee*
Knoxville, TN 37996-1301 USA
Ad Emmen *SARA*
P.O. Box 4613, 1009 AP Amsterdam, The Netherlands
Jaap Hollenberg *SARA*
P.O. Box 4613, 1009 AP Amsterdam, The Netherlands
Yasumasa Kanada *University of Tokyo*
2-11-16 Yayoi Bunko-ku, Tokyo, Japan
Rossend LLurba *TUD*
P.O. Box 354, 2600 AJ Delft, The Netherlands
Raul Mendez *ISR*
8/F Recruit Kachidoki Bldg.,
2-11 Kachidoki, Chuo-ku, Tokyo 104, Japan
Aad van der Steen *ACCU*
P.O. Box 80011, 3508 TA Utrecht, The Netherlands

Printed by Wim Schroot, Amsterdam.

Design by Van Dijk/Eger/ Associates, Zeist.

SUPERCOMPUTER is published by ASFRA bv, Edam, The Netherlands, in association with SARA, Amsterdam and Stichting Supercomputing Support Services (4-S).

ISSN 0168–7875

SUPERCOMPUTER

Contents

Contents

Parallelization of a multigrid solver on the KSR1

Ulrich Schwardmann

Gesellschaft für wissenschaftliche Datenverarbeitung Göttingen, Am Faßberg, D-37077 Göttingen, Germany

©

Supercomputer **55**, X–3
Received August 1993

The multigrid algorithm is known as a fast method for the solution of differential equations with boundary conditions. For the temporal simulation of physical systems for instance, where an accurate solution is needed for each time step, a fast code, handling big data arrays, is especially necessary. The goal here is the parallelization and optimization of the multigrid Poisson solver for a parallel computer with modern virtual shared memory concept.

The Multigrid Poisson Solver

The Poisson equation

$$L^{\Omega}u := -\Delta u = f(z), z \in \Omega$$

on a given domain Ω with Dirichlet conditions on the boundary $\partial\Omega$

$$L^{\partial\Omega}u := u = g(z), z \in \partial\Omega$$

is a famous paradigm of the elliptic boundary value problem. The multigrid method is applied here to solve it as a model problem. To control the convergence behavior of the algorithm, a boundary condition problem with known solution is used with the unit square as the domain Ω.
The problem is discretized by replacing Ω with a discrete uniform square grid Ω_h of mesh size h. A five-point difference scheme approximates the Laplace operator Δ. The resulting sparse linear equations $L_h u_h = f_h$ can then be solved with the relaxation formula

$$u^h_{i,j} := \frac{1}{4}h^2(f^h_{i,j} + u^h_{i,j-1} + u^h_{i-1,j} + u^h_{i+1,j} + u^h_{i,j+1})$$

by the multigrid iteration and the nested iteration based on approximations on coarser grids, as explained in detail by [1] and [2]. The mesh sizes for the coarser grids used here are $h_k = 2^{n-k}h$ on the sequence of grids $\{\Omega_{h_k}\}_{1\le k\le n}$, where $\Omega_h = \Omega_{h_n}$ with $(2^n+1)^2$ grid points. The grid Ω_{h_k} will indicate the grid level $n-k$ in the following. The code used here is based on the algorithm published in [2], using red-black relaxation.

The architecture and the programing model

The cells (nodes) of the parallel computer KSR1 have a superscalar 64-bit processor, a local cache of 32 Mbyte and fast local subcaches for data and instructions, both with 256 Kbyte. The configuration used consists of 32 such cells, connected by a fast unit-direction ring. The view of the memory is an uniform adress space defined as a virtual shared memory in a three-level hierarchy: the local subcache, the local cache and the union of all caches in the ring. The 32 cells of the configuration used are partioned into sets of cells (called Psets). All tests were made with a maximum of 16 cells.
The programming model of the KSR1 is characterized by its virtual shared memory. Parallelism can be defined on different code segments as well as on different parts of the data structure. Tiling is the most convenient way to describe data parallelism, and was used here whereever possible.

Global timing analysis of the multigrid Poisson solver

After implementing the multigrid algorithm in its serial form for a grid of level 10 (i.e. $(2^{10}+1)^2$ grid points) on the KSR1 we tested its timing behavior using the Unix profiling tool `gprof`.

Granularity: each sample hit covers 8 byte(s) for 0.02% of 130.22 seconds

% time	Cumulative seconds	Self seconds	Calls	Self ms/call	Total ms/call	Name
25.0	32.54	32.54				`cos [5]`
23.1	62.60	30.06	400	75.15	75.15	`relaxs_ [6]`
12.4	78.78	16.18	4	4045.00	5050.00	`diff_ [7]`
10.4	92.26	13.48	36	374.44	378.89	`int4_ [8]`
5.3	99.22	6.96	4	1740.00	19089.98	`mgpois_ [3]`
5.0	105.78	6.56	4100	1.60	1.60	`f_ [10]`
4.0	110.96	5.18	4	1295.00	3045.00	`init1_ [9]`
3.7	115.76	4.80	180	26.67	26.67	`restr_ [11]`
3.3	120.10	4.34	36	120.56	120.56	`trans_ [12]`
3.3	124.42	4.32	180	24.00	24.00	`int2a_ [14]`
3.1	128.44	4.02	4100	0.98	0.98	`exsol_ [15]`

Figure 1. Extract of the global computing time analysis for the multigrid program by `gprof`.

As can be seen in Figure 1, beside the functions `cos` and `diff`, which are used here only to verify the correctness of the program, most of the time (without `cos` and `diff` more than a third) is spent in the relaxation. Roughly half of this time is spend in the 4th order interpolation, that brings the starting approximation to the finest grid of the cycle. The restriction and bilinear interpolation follow after some driver and initialization routines and take again about a third of the time.
In order to get an even more accurate analysis for these subroutines, it was necessary to measure the time spent in each cycle and on each grid level. This was done by using the KSR1 monitor function `usertime`.

The timing behavior of these subroutines is approximately linear to the number of grid points. The time spent in each cycle is dominated by the time spent for relaxation on the finest grid, which is about a half of the time of the whole cycle. Therefore the optimization and parallelization has to be done first on the finer grids and especially for the relaxation on these grids.

Parallelization of the Multigrid Poisson Solver

Automatic parallelization by KAP. The first step in our process of parallelization was to use the automatic parallelization facility of the Fortran preprocessor KAP and to run the code parallel on four cells.

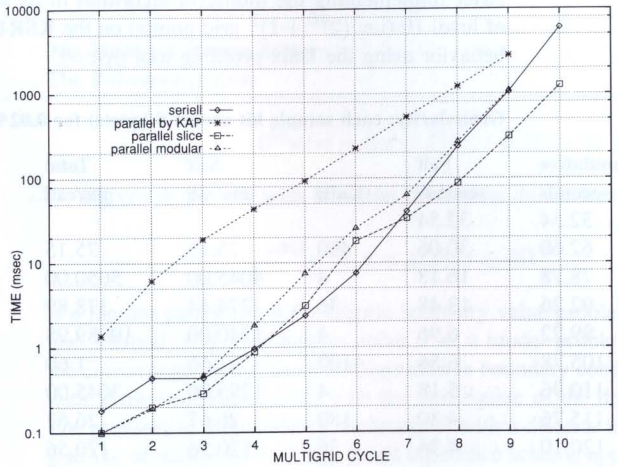


Figure 2. Comparison of the computing time for whole multigrid cycles between the serial case, the automatic parallelization by KAP, the parallelization along columns on four cells with four threads (slice) and sixteen threads (modular).

Because of data dependencies KAP decides to parallelize the relaxation along the rows of the grid. But to compute the five-point scheme of the relaxation, this means that the communication, that takes place on the boundaries of the tiles, becomes very expensive. As shown in Figure 2 this leads to an increase of computing time of the whole program.

Because the unit of fequat communication on the KSR1 is a subpage of 128 bytes, each entry-point on the boundary has to be sent with its complete subpage. With a tiling along the rows this is a small column of 16 entry points in the grid, which lies rectangular to the tile boundary.

Parallelization of the relaxation. In order to avoid this communication overhead, one has to divide the grid along the columns. Only with a large

number of tiles (≥ 32) the number of boundary cells in a complete division along columns increases in such a way, that a rectangular division of the grid will become significantly more efficient.

To divide the grids along its columns, it is necessary to make some changes in the original algorithm of the multigrid solver as published in [2].

The red-black relaxation used here steps in a chess board manner through the grid. This means that it goes twice with stride two through the complete data. The starting point of computation in the column is controlled in the original algorithm by a global variable that depends only implicitly on these parameters and is changed after each computation on a column. For the parallelization of the code one has to make these dependencies explicit and has to use private variables to control the starting points.

During relaxation on a point $u_{i,j}$ one computes the diagonal sums

$$diag_{i,j}^{(ul)} = u_{i,j-1} + u_{i-1,j} \text{ and } diag_{i,j}^{(lr)} = u_{i+1,j} + u_{i,j+1}$$

which implies $diag_{i+1,j+1}^{(ul)} = diag_{i,j}^{(lr)}$.

In the original algorithm these diagonal sums are computed once for each grid column, stored into an array and used twice to avoid redundant operations. This reduces the number of computations, but increases the number of memory calls, which can be a disadvantage when memory bandwidth is a constraint. Moreover, this precomputation increases the data dependences and is therefore not used here.

After implementing these changes the parallelization of the relaxation was done by dividing the grids with sufficiently many grid points along columns into tiles. Without tiling the data in the other routines this did not lead to an essential decrease of computing time compared to the serial code.

In the cycles in the medium range there is even an increase (see Figure 2). This can be explained by the overhead that results from the data distribution to the four cells, which has to be done between each relaxation and interpolation or restriction. Therefore an efficient parallelization had to be done coherently for all parts of the cycle.

Parallelization of the full multigrid method. The parallization of the multigrid method by tiling the data space is possible only if there are enough columns of the grid to be distributed to the cells. The multigrid cycle has therefore to be performed in a hybrid way, with serial code for the coarser grids and parallel code for the finer grids.

The parallel relaxation was used here only for grid levels greater or equal to six, the parallel interpolation and restriction between grid levels five and six and higher. This proved to be a good compromise between the data distribution cost, when parallel restriction and interpolation needs data from grids relaxed serially, and the parallelization overhead on coarser grids.

In Figure 2 one sees the computing time for the parallelization of whole multigrid cycles computed on four cells compared to the serial code. One

curve shows the use of the tiling strategy *slice*, that means a one to one correspondence between tiles and threads. Another shows the use of the tiling strategy *modular* where sixteen tiles are mapped onto four threads working on the four cells used.
In the first case one sees a significant reduction of computing time on the finer grids.
As shown in Figure 2, in the medium range the computation time with the modular tiling strategy is even significantly more than the serial code. In this case there is no reuse of data stored in the subcache, because data of completely different parts of the grid are used on one cell and, after computing one tile, the data of the other tiles is lost. Furthermore, the amount of boundary points, where the values has to be communicated between threads, is increased. Because the threads are mapped onto the cells in a modular way there is an increase of communication overhead whereas the degree of parallelization is the same as before. This strategy is therefore avoided in our further optimizations.

Speedup and efficiency of the parallelization. The parallelization of the multigrid solver with slice strategy was then tested on a varying number of cells and the timing results of these tests for the most significant parts of the algorithm were compared with the timings of the serial code. The resulting speedup by parallelization, the quotient of the serial by the parallel time, is shown in Figure 3. Remarkable here is the *superlinear* speedup, that is most significant for the relaxation on the grid of level 10.
This speedup can be explained by the division of the data space into parts distributed onto different cells, where these smaller parts can be hold by memory elements in higher levels of the memory hierarchy of the system. In this case the amount of data exceeds 20 Mbyte, and therefore does not fit into the cache of one cell.
This leads to an efficiency of more than $5/4$ for the relaxation on the finest grid on more than two cells. But even the efficiency of the 4^{th} order interpolation exceeds $9/10$ and the efficiency of the computation over all cycles is about $3/4$.

Optimization of the relaxation

In order to further increase the performance of the multigrid algorithm we performed some experiments in optimizing the most time-consuming part of the algorithm, the relaxation. Subpage alignment and manual loop unrolling did not lead to any significant improvement, and, with the change of the data structure of the grid array into a field, efficient for most vector computers at least on the medium grid levels (see [3] and [4]), the algorithm even took longer.
But the improvement of the locality of data turned out to be the key to get better performance. As mentioned earlier, for each call the red-black relaxation steps twice through the whole data area with stride two. On finer grids this means that all data have to be moved twice into the subcache.

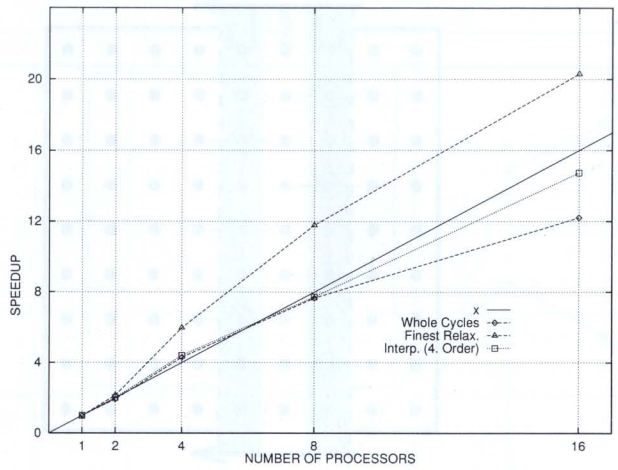


Figure 3. Speedup of the parallelization along columns on a various number of processors for all cycles up to level 10, and for the 4^{th} order interpolation and the relaxation on level 10.

A better re-use of data in the subcache can be achieved by using a macro pipeline on the columns of the grid. The relaxation of the red points of a column, followed by a delay in order to compute all red points needed for the last step in which the black points are relaxated (see Figure 4). The data going through this macro pipeline must be loaded for the first step only once with stride two into the subcache. For grids up to level 11 the column still fits into the subcache at the third step.
This macro pipeline code has to be parallelized manually because the pipeline must be started and stopped accurately at the segment boundaries. This cannot be controlled with the tiling parameters.
A comparison of the two algorithms performed on 16 cells and on the different grid levels up to level 11 is given in Figure 5. This figure shows the advantage of the macro pipeline on all grid levels greater then 8, but also in the area of serial execution for grid levels less then 6. At grid level 6 one sees again the switching point between serial and parallel code, where communication decreases the performance.
At the grid levels 7 and 8 the parallel performance of the macro pipeline code is lower than the original parallel relaxation. This is due to the starting time of the macro pipeline, which is high on these grid levels compared to the number of computed columns in each tile.
The difference between the two methods is most significant at grid level 10. Here one sees already a clear decrease of performance for the original parallel relaxation. The data, when distributed over 16 cells, still just fits

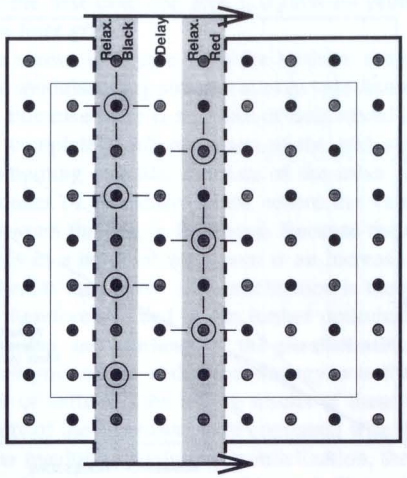


Figure 4. The Macro Pipeline stepping through a grid of level 3.

into the subcache on grid level 9, and can therefore be reused in the second relaxation step. But at grid level 10 the whole subcache must be reloaded. The optimized relaxation with macro pipeline on the other hand avoids this reloading and therefore the performance still increases at this grid level. This leads for this algorithm on 16 cells to a speedup of more than 34 compared to the original code.

At grid level 11 the size of one column already exceeds one page. Therefore all computations next to the borderlines of the tiles invalidate large areas of the concerned pages. This leads to an substantial increase of page movement, measured by the KSR routines for parallel monitoring (`pmon_delta`). The performance decreases here for both algorithms by about 20 Mflop/s.

The macro pipeline algorithm was tested on a grid of level 12 as well. Here the amount of data needed in the macro pipeline is more than a half of the subcache. So the random replacement rule for the subcache management destroys with high probability parts of the macro pipeline in the subcache. Furthermore the complete data for the grids on level 12 exceed a quarter Gbyte. Parts of these data must be held on several nodes at the same time, with the effect that the grids do not fit into the user area of the used caches anymore. But this situation is still managed by the operating system (Rel. 1.1.3), even if the performance decreases to 50 Mflop/s, and the amount of subcache-misses, subpage-misses on cache and page-misses increases dramatically. A tiling into two times eight

rectangles did not improve the performance. The communication along the horizontal borderlines took too much time and, because of more data duplicity, the number of subpage-misses increased. On a larger number of cells (32) with a tiling into two times sixteen rectangles these problems should vanish.

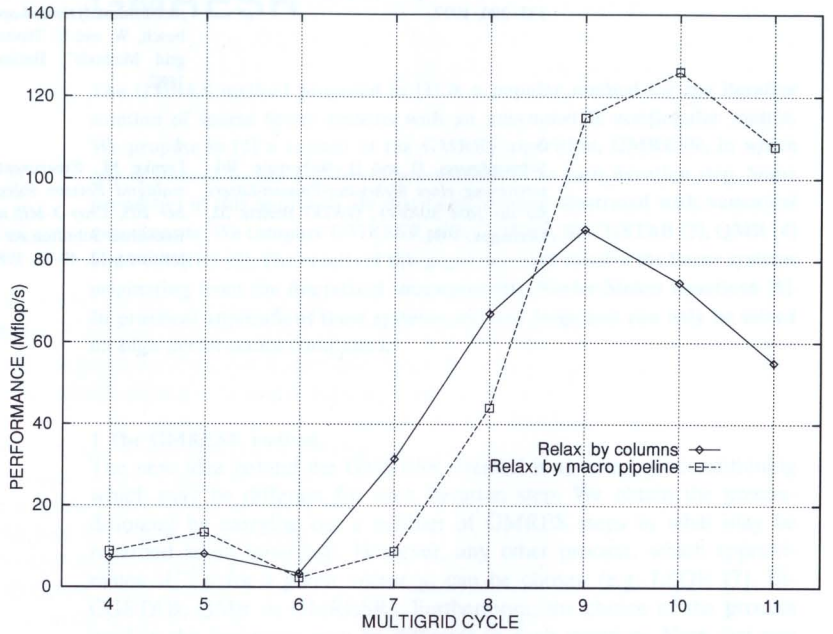


Figure 5. Performance on 16 cells of the parallel relaxation on the various grid levels (up to grid level 10), unoptimized version versus optimized version by macro pipelining.

Conclusion

Because of its complex memory hierarchy, the key to achieve high performance on the parallel computer KSR1 is the optimization of data locality.

Although the relaxation dominates the computing time, this means firstly, that to minimize communication costs *all* participating subroutines must be included into the parallelization process of the multigrid method. Proceeding in this way a speedup of 11.50 for the full multigrid method on 16 cells is obtained. This leads to an overall efficiency of about 0.72.

By improving data locality inside the relaxation algorithm a significantly higher speedup of 13.27 for the whole multigrid algorithm can be achieved.

Because of the linear increase of memory with the number of cells it is possible to compute solutions on much finer grids. On a machine with 64 cells, for instance, the multigrid algorithm should work on a grid of

$(2^{13}+1)^2$ grid points, tiled into four times sixteen rectangles, with good performance within a reasonable time.

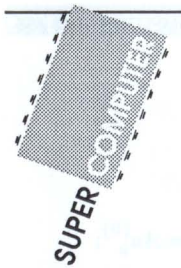
References

1
Brandt, A., *Multi-level adaptive solutions to boundary-value problems*, Math. Comp. **31**, 333–390, 1977.

2
Stüben, K. and U. Trottenberg, *Multigrid methods: Fundamental algorithms, model problem analysis and applications*, In: Hackbusch, W. and U. Trottenberg (eds.) "Multigrid Methods", Berlin, Springer, 1–176, 1982.

3
Schwardmann, U and H. Weberpals, *Vektorisierung eines Mehrgitter-Poisson-Lösers für die IBM 3090/VF*, GWDG Bericht **32**, Göttingen, 1991.

4
Lemke, M., *Experiments with a vectorized multigrid Poisson solver on the CDC Cyber 205, Cray X-MP and Fujitsu VP 200*, Bochumer Schriften zur Parallelen Datenverarbeitung **12**, 49-85, 1987.



Contributions

Further experiences with GMRESR

C. Vuik

Delft University of Technology, Faculty of Technical Mathematics and Informatics, PO Box 5031, NL-2600 GA Delft, The Netherlands

©
Supercomputer **55**, X–3
Received July 1993

The GMRES method proposed in [1] is a popular method for the iterative solution of sparse linear systems with an unsymmetric nonsingular matrix. We propose in [2] a variant of the GMRES algorithm, GMRESR, in which one is allowed to use a different preconditioner in each iteration step. Some properties of this approach are discussed hereand illustrated with numerical experiments. We compare GMRESR with GMRES, Bi-CGSTAB [3], QMR [4] and FGMRES [5]. The results of this paper are used to solve the linear systems originating from the discretized incompressible Navier-Stokes equations [6]. In practical applications these systems are very large and can only be solved on high-performance computers.

1 The GMRESR method

The new idea behind the GMRESR method is to use a preconditioning which may be different for each iteration step. We obtain the preconditioning by carrying out a number of GMRES steps in what may be regarded as an innerloop. However, any other process, which approximates $A^{-1}y$ for a given vector y, can be chosen (e.g. LSQR [7], Bi-CGSTAB, QMR or GMRESR). Furthermore, the choice of the process used in the innerloop may be different in each iteration. Note that one can use GMRESR in a recursive way, which motivates the name of the method GMRESR(ecursive).

We denote the approximate solution of $A^{-1}r$ by $P_m(A)r$, where P_m represents the GMRES polynomial that is implicitly constructed in m iteration steps of GMRES. Note that this polynomial depends on the residual r, so that we have effectively different polynomials in different steps of the outer iteration. We will make this dependence explicit by adding the number of the current outer iteration as an index to $P : P_{m,k}(A)r_k$. The resulting process, GMRESR, is represented by the following iteration scheme for the solution of $Ax = b$:

GMRESR algorithm

1. Start: Select x_0, m, ϵ;
   $r_0 = b - Ax_0, \quad k = -1;$
2. Iterate: **while** $\|r_{k+1}\|_2 > \epsilon$ **do**
   $k = k + 1, \quad u_k^{(0)} = P_{m,k}(A)r_k, \quad c_k^{(0)} = Au_k^{(0)};$
   for $i = 0, ..., k-1$ **do**
   $\alpha_i = c_i^T c_k^{(i)}, \quad c_k^{(i+1)} = c_k^{(i)} - \alpha_i c_i,$
   $u_k^{(i+1)} = u_k^{(i)} - \alpha_i u_i;$
   endfor
   $c_k = c_k^{(k)}/\|c_k^{(k)}\|_2; \quad u_k = u_k^{(k)}/\|c_k^{(k)}\|_2;$
   $x_{k+1} = x_k + u_k c_k^T r_k;$
   $r_{k+1} = r_k - c_k c_k^T r_k;$
   endwhile

In the remainder of this paper, the process to calculate $u_k^{(0)}$ is called the innerloop of the GMRESR method. If GMRES in the innerloop stagnates we obtain $u_k^{(0)} = 0$ and the method breaks down. In such a case we avoid break down by using one step of LSQR [7] in the innerloop: $u_k^{(0)} = A^T r_k$. In Section 6 we give a motivation for this choice and specify some examples where the LSQR switch gives a much better convergence behavior.

A more practical scheme, in our opinion, arises when the outer iteration is restarted after l_s iterations, in order to limit memory requirements, or to include only updates from the last l_t outer iterations (a truncated GMRESR version). The resulting scheme is denoted by GMRESR (l_s, l_t, m). In Section 3 we give other truncation strategies and compare restarting and truncation for some testproblems. The GMRESR method without restarting or truncation is denoted by GMRESR(m).

In [2] it is shown that GMRESR(m) is a robust method, and that it is a minimal residual method. For other properties of GMRESR we refer to [2].

The choice of m. In order to compare the efficiency of GMRES and GMRESR(m), estimates for the amount of work and the required memory of both methods are listed in Figure 1.

Figure 1. Amount of work and memory for GMRES and GMRESR(m).

Method	GMRES	GMRESR(m)
steps	m_g	m_{gr}
matvec	m_g	$m_{gr} \cdot m$
vector updates	$\frac{1}{2}m_g^2$	$m_{gr} \cdot (\frac{m^2}{2} + m_{gr})$
inner products	$\frac{1}{2}m_g^2$	$m_{gr} \cdot (\frac{m^2}{2} + \frac{m_{gr}}{2})$
memory vectors	m_g	$2m_{gr} + m$

It appears from our numerical experiments that $m_{gr} \cdot m$ is in many cases approximately equal to m_g. This observation is used to derive

optimal choices for m with respect to work and required memory. In the following, we assume that a vector update and inner product have the same computational costs. Using $m_{gr} = m_g/m$ it appears that the minimal amount of work is attained for $m = \sqrt[3]{3m_g}$, and it is less than $2.5\ m_g^{4/3}$. Note that the amount of work of GMRES is equal to m_g^2. With respect to memory requirements the optimal value is equal to $m = \sqrt{2m_g}$, so the amount of memory for GMRESR(m) is equal to $2\sqrt{2m_g}$. This combined with Figure 1 implies that for large values of m_g, GMRESR(m) needs much less memory than GMRES. Both optimal values of m are slowly varying with m_g. Thus a given m is near-optimal for a wide range of values of m_g. Note that the optimal m with respect to work is in general less than the optimal m with respect to memory. It depends on the problem and the available computer, which value is preferred. In our experiments we observe that for both choices the amount of work and required memory is much less than for GMRES.

2 Numerical experiments

In this section we illustrate the theoretical properties of GMRESR using numerical experiments where we use a linear system obtained from a discretization of the following PDE:

$$-(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) + \beta(\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}) = f \text{ on } \Omega,$$

$$u|_{\partial\Omega} = 0,$$

where Ω is the unit square. The exact solution u is given by $u(x, y) = sin(\pi x)\, sin(\pi y)$. In the discretization we use the standard five-point central finite-difference approximation. The stepsizes in x- and y-direction are equal. We use the following iterative methods: GMRES(m), Bi-CGSTAB, and GMRESR(m). We use a more or less optimal choice of m to obtain results using GMRES(m). We start with $x_0 = 0$ and stop if $\|r_k\|_2/\|r_0\|_2 \leq 10^{-12}$.

In Figure 2 we present results for GMRESR(m). CPU time is measured in seconds using 1 processor of a Convex C-3820. Using full GMRES we observe that $m_g = 183$, which means that one needs 183 vectors in memory. Furthermore full GMRES costs 4.4 seconds CPU time. Note that $m_{gr} \cdot m$ is for small values of m approximately equal to m_g. Suppose m_g is unknown and we use $m_{gr} \cdot m$ as an approximation of m_g. Then using the formulas given in Section 1, we obtain the following bounds for the optimal values of m:

work: $\sqrt[3]{3 * 188} = 8.3 < m < 9.4 = \sqrt[3]{3 * 280}$,
memory: $\sqrt{2 * 188} = 19.4 < m < 23.7 = \sqrt{2 * 280}$.

Comparing this with Figure 2 we note that there is a good correspondence between theory and experiments in this example. As expected, the optimal value of m with respect to memory, is larger than with respect

m	4	8	12	16	20
m_{gr}	47	25	19	16	14
$m \cdot m_{gr}$	188	200	228	256	280
CPU time (s)	0.82	0.57	0.68	0.83	1.01
memory vectors	98	58	50	48	48

Figure 2. The results for GMRESR(m) with β=1 and h=1/50.

to work. However, for both choices of m we observe a considerable gain in computing time and memory requirements.
Results comparing the three iterative methods are shown in Figure 3.

Method	Iterations	matvec	CPU time (s)
GMRES(32)	1355	1355	26
Bi-CGSTAB	237	474	1.7
GMRESR(10)	36	360	4.3

Figure 3. Results for β=1 and h=1/100.

It appears that GMRESR(10) is better than GMRES(32). Although Bi-CGSTAB uses less CPU time, it uses more matrix-vector products than GMRESR(10).
In Figure 4 we take the stepsize $h = 1/100$ and β a function of x and y as follows:

$$\beta(x,y) = \begin{cases} 1 & \text{for } x,y \in [\frac{1}{2},\frac{3}{5}]^2 \\ 1000 & \text{for } x,y \in [0,1]^2 \setminus [\frac{1}{2},\frac{3}{5}]^2 \end{cases} \quad (1)$$

Method	Iterations	Matvec	CPU time (s)
GMRES(32)	1536	1536	30
Bi-CGSTAB	no convergence		
GMRESR(10)	56	560	7.8

Figure 4. Results, with β given by (1) and h=1/100.

Note that in this problem GMRESR(10) is the best method.
The following example comes from a discretization of the incompressible Navier-Stokes equations. This discretization leads to two different linear systems, the momentum equations and the pressure equation (for a further description we refer to [2, 6]). We consider a specific testproblem, which describes the flow through a curved channel. The problem is discretized with 32×128 finite volumes. The pressure equation is solved using an average of an ILU and MILU preconditioner with $\alpha = 0.975$ [6, 8]. We start with $x_0 = 0$ and stop when $\|r_k\|_2/\|r_0\|_2 \leq 10^{-6}$. The results are shown in Figure 5. Note that for this problem GMRESR(4) is the fastest method with respect to CPU time.

Method	Iterations	Matvec	CPU time (s)
full GMRES	47	47	0.77
CGS	38	76	0.49
Bi-CGSTAB	34	68	0.44
GMRESR(4)	12	48	0.43

Figure 5. Iterative methods applied to the pressure equation.

3 Restarting and truncation strategies

We present some truncation strategies and compare the results with restarted and full GMRESR.
There are many different ways to truncate GMRESR. The first one, which is already given above, is to use the l_t last search direction (denoted by *trunclast*). To obtain another truncation strategy we note that in many cases GMRESR has a superlinear convergence behavior. This means that after some iterations GMRESR converges as if some eigenvalues of the matrix are absent (compare [9]). Restarting or truncation can destroy this behavior ([10]; pp. 1334–1335). If superlinear convergence occurs, it appears a good idea to use the $l_t - 1$ first search directions and 1 last search directions (denoted by *truncfirst*). Both strategies are used in the following experiments. Figure 6 shows that truncation with $l_t - 1$ first and 1 last search directions is the best strategy for this example. If there are only 18 memory vectors available, the gain in CPU time for $l_s = 50, l_t = 5$ with respect to $_s = l_t = 5$ is equal to 40%. Furthermore, comparing full GMRESR(8) with GMRESR(50,10,8) (*truncfirst* variant) we see that the amounts of CPU time are approximately the same, whereas the amount of memory is halved.

$l_t = l_s$	Restart		$ls = 50$	*trunclast*		*truncfirst*		Memory
	Iterations	CPU	l_t	Iterations	CPU	Iterations	CPU	Vectors
5	57	1.15	5	41	0.87	37	0.79	18
10	45	0.97	10	32	0.73	29	0.68	28
15	33	0.74	15	29	0.69	26	0.62	38
20	29	0.67	20	25	0.60	25	0.60	48
25	25	0.60	25	25	0.60	25	0.60	58

Figure 6. Results with GMRESR(l_s,l_t,8),β=1 and h=1/50, CPU time in seconds.

We conclude this section with some other truncation strategies. First we note that it seems an awkward choice to use one last search direction in the *truncfirst* variant. This choice is motivated by the fact that if one applies GCR to a symmetric problem then it is necessary and sufficient to use one last search direction in order to obtain the same convergence behavior as full GCR. We have done experiments without this final direction (*truncfirst1*). These results are given in Figure 7. Note that the *truncfirst1* variant is the worst truncation strategy, so it is indeed a good idea to include one last search direction, which is done in the original *truncfirst* variant.
Finally in ([10]; p. 1335) another truncation strategy is proposed for a GCR-like method. For the GMRESR algorithm this strategy leads to the

l_t	5	10	15	20	25
truncfirst1	55	49	34	26	25
minalfa	36	28	25	25	25

Figure 7. Number of iterations for GMRESR(50,l_t,8), β=1 and h=1/50.

following variant (*minalfa*): if $k \geq l_t$ then the search direction with the smallest absolute value of α_i in the **for**-loop is thrown away. The motivation is that the search direction with the smallest $|\alpha_i|$ has only a limited influence on the GMRESR convergence. From Figure 7 it appears that this leads to the best truncation strategy for this example. Another important advantage of the *minalfa* variant is that it is a black box strategy. For instance if the bad eigenvector components (with respect to the convergence behavior) appear after some iterations, then the *truncfirst* variant is much worse than the *minalfa* variant.

4 Some ideas for choosing an iterative solution method.

There are a large number of known iterative solution methods for non-symmetric problems. In this section we present some ideas to motivate a choice of a feasible iterative method. These ideas are based on our experiments. Probably they should be adapted for other classes of problems. The insights in this section can be used to guess a priori if it has sense to change from one iterative method to another. Furthermore it is shown that two parameters: the ratio of the CPU time for a matrix-vector product and a vector update, and the expected number of full GMRES iterations, are important to choose an iterative method. Finally, the ideas given in this section show a good agreement with our experiments given above.

In the remainder of this section we assume that the amount of required memory is available. Otherwise restarted or truncated versions of GMRES (GMRESR) can be used, however, it is not clear if the results in this section holds in such a case.

The CPU time of many iterative methods consists of two main parts:

- the total CPU time used for matrix-vector products, which is denoted by t_m (if a preconditioner is used, t_m includes the time for preconditioning);
- the total CPU time used for vector updates and inner products, which is denoted by t_v.

The total time t_m for GMRES (GMRESR) is always less than the time t_m for other iterative methods. On the other hand, the time t_v for GMRES (GMRESR) can become very large if the number of iterations increases. We prefer GMRES (GMRESR) if the ratio $\gamma = t_v/t_m$ is not too large (≤ 0.5 in this paper). Note that for every other Krylov subspace method the gain in CPU time is always less than $\frac{\gamma}{1+\gamma} \cdot 100\%$. In our experiments it appears that for the choice $\gamma = 0.5, (1+\gamma)t_m$ of GMRES (GMRESR) is approximately equal to $t_m + t_v$ of Bi-CGSTAB. So in our example at the end of this section we take $\gamma = 0.5$ (for this choice $\frac{\gamma}{1+\gamma} \cdot 100\% = 33\%$).

The CPU time used for one matrix (+ preconditioner) vector product is denoted by t_{m1}, and the CPU time of one vectorupdate (or inner product) is denoted by t_{v1}. The factor f is defined by $f = t_{m1}/t_{v1}$. Using the assumption that $m_{gr} \cdot m \cong m_g$ we obtain the following expressions:

GMRES: $t_m = m_g \cdot t_{m1}, \quad t_v = m_g^2 t_{v1}$;

GMRESR: $t_m = m_g \cdot t_{m1}, \quad t_v = 2.5 m_g^{4/3} t_{v1}$.

As said before we prefer GMRES (GMRESR) if $t_v \leq \gamma t_m$. For GMRES this means $m_g \leq \gamma f$, whereas for GMRESR this means $m_g \leq (\gamma f/2.5)^3$. For GMRESR the ratio $t_v/t_m = 2.5(m_g)^{1/3} t_{v1}/t_{m1}$ is a slowly varying function of m_g.

Figure 8 illustrates the given bounds for the choice $\gamma = 0.5$. We emphasize that this figure only gives qualitative information. It illustrates the dependence of the choice on f and m_g. Below we specify some applications using this information.

- For a given system and computer, f can be measured. This together with an estimate of m_g and Figure 8 gives an impression of which iterative method is feasible.
- Suppose Bi-CGSTAB is the best method for a certain problem, without preconditioning. Including preconditioning, Figure 8 suggests that GMRESR can be better for this preconditioned system, because m_g is (much) lower and f is (much) higher (in general a preconditioner is harder to vectorize than a matrix-vector product).

Note that the applicability of GMRESR for large values of f is much wider than GMRES.

For the first example given in Section 3, $f = 10$, so Figure 8 agrees with our observation that Bi-CGSTAB costs more matrix-vector products but less CPU time than GMRESR. In the practical examples given in Section 3 and [2], $f = 20$ and $m_g \leq 50$. In these examples the CPU time of GMRESR is less than the CPU time of Bi-CGSTAB, which is also in accordance with Figure 8.

Finally we compare GMRESR with QMR. It is easily seen from [4], equations (2.7), (2.8) and (3.1) that the QMR method uses k multiplications with A and A^T to construct a solution, which is an element of a Krylov subspace with dimension k. So we choose $\gamma = 1$ in order to compare GMRESR and QMR (Figure 9). From Figure 9 we note that GMRESR has a large region of feasibility with respect to QMR.

5 Comparison of GMRESR with FGMRES

Another GMRES-like iteration scheme with a variable preconditioner is proposed in [5]. In Saad's scheme (FGMRES) a Krylov subspace is generated that is different from ours (GMRESR). We specify an example for which FGMRES breaks down. Comparison shows that in our examples the convergence behavior of GMRESR and FGMRES are approximately the same. An advantage of GMRESR is that it can be truncated and/or restarted, whereas FGMRES can only be restarted. Above we have seen that in some problems, truncated GMRESR converges faster

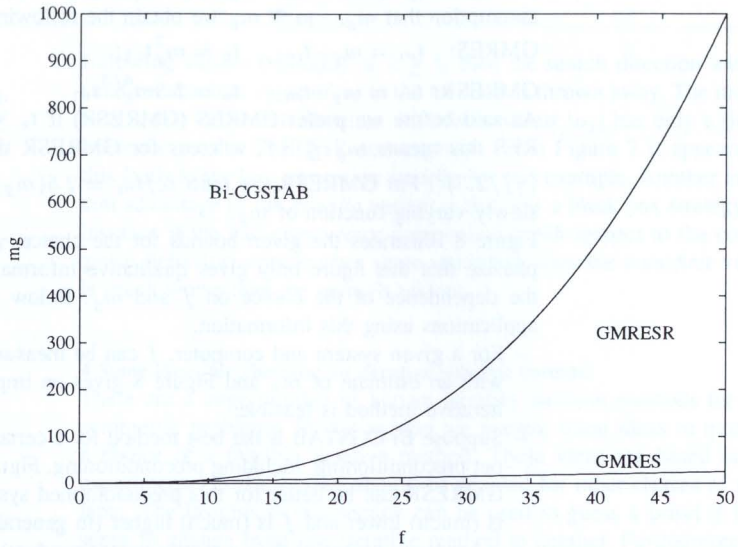


Figure 8. Regions of feasibility of Bi-CGSTAB, GMRES, and GMRESR for γ=0.5.

than restarted GMRESR. Using such an example we show that restarted FGMRES costs more CPU time than truncated GMRESR.

A well-known property of GMRES is, that it has no serious breakdown. From the following example we conclude that it is possible that FGMRES breaks down. For the algorithm we refer to ([5]; p.4) and note that FGMRES is equal to GMRES if $M_j = M$ for all j. So breakdown of FGMRES is only possible if one chooses different M_j.

Example.

Take $A = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$, $x = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ and $x_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$. In Algorithm 2.2 of [5] we choose $M_1 = I$ and $M_2 = A^2$. These choices lead to $z_1 = z_2 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, $v_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, $v_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$, and $h_{3,2} = 0$, which implies that v_3 does not exist. Since $x_2 = x_0 + \alpha z_1 + \beta z_2$, it follows that $x_2 \neq x$, so this is a serious breakdown.

In the innerloop of GMRESR, we calculate an approximate solution of $Au_k^{(0)} = r_k$. In FGMRES an approximation of $Az_k = v_k$ is calculated. If the preconditioner is the same for every k, then u_i and z_i, $i = 0, \ldots, k$ span the same Krylov subspace. However, if the preconditioner varies,

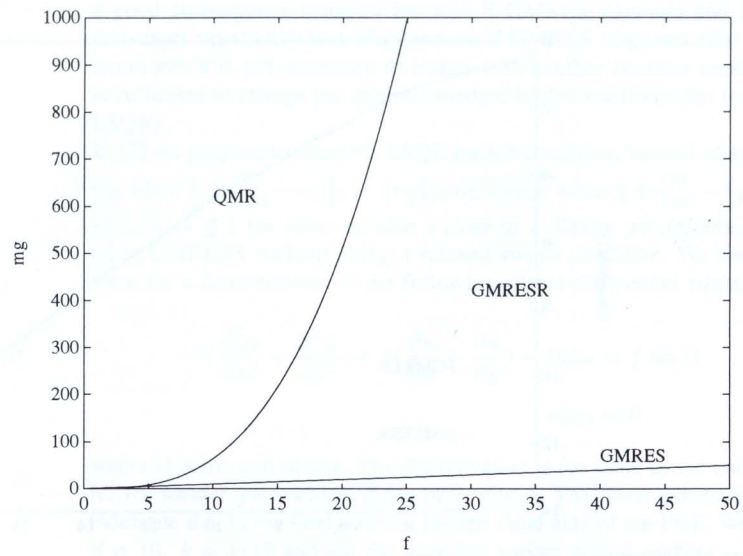


Figure 9. Regions of feasibility of QMR, GMRES, and GMRESR for γ=1.

the Krylov subspaces can be different. To illustrate this we calculate the solution of the problem given in section 3, with $\beta = 1$ and $h = 1/50$. As innerloop we take one step of GMRES(10) in both methods. The results are given in Figure 10. As expected $\|r_o\|_2$ and $\|r_1\|_2$ are the same for both methods. In this example the differences of the norms of the residuals are small. We have also done experiments with the same search directions in both methods. In these experiments the results of GMRESR and FGMRES are the same to machine precision.

It follows from Algorithm 2.2 [5] that only the vectors v_k are updated in the orthogonalization process. Assuming that GMRESR and FGMRES use both m_{gr} iterations for convergence, GMRESR needs $\frac{1}{2}m_{gr}^2$ vector updates more than FGMRES. Note that GMRESR and FGMRES are feasible for relatively large values of f (see section 4 for the definition of f). In this case the CPU time of $\frac{1}{2}m_{gr}^2$ extra vectorupdates is negligible. Finally we compare restarted FGMRES and truncated GMRESR. As we already note, it is impossible to truncate FGMRES. In Figure 11 we give results for both methods, for $\beta = 1$ and $h = 1/100$. As innerloop we use one step of GMRES(10) for both methods. For this example FGMRES is more expensive than GMRESR. If there is only a small number of memory vectors available ($\leq$ 20), then FGMRES uses two times as many iterations and two times as much CPU-time.

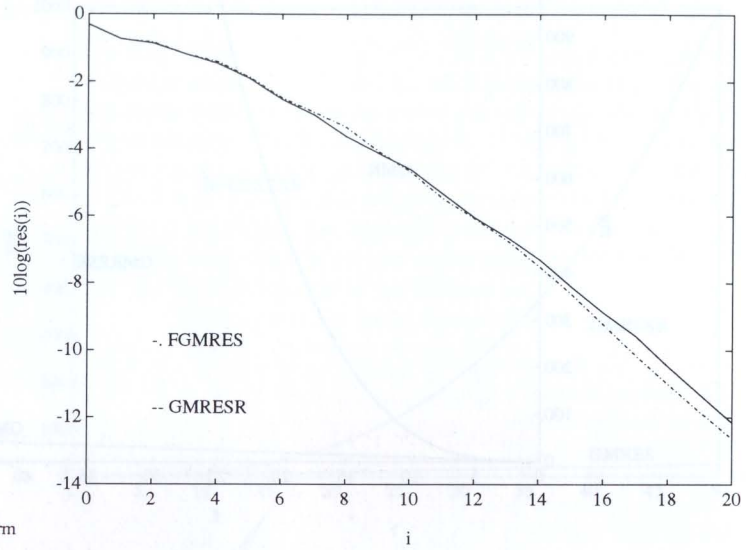


Figure 10. The norm of the residuals for β=1 and h=1/50.

	FGMRES			GMRESR (*truncfirst*)		Memory
l_s	Iterations	CPU	l_t	Iterations	CPU	vectors
5	128	12.3	5	64	6.5	20
10	83	8.2	10	46	4.9	30
15	68	6.9	15	41	4.7	40
20	59	6.1	20	41	4.8	50
25	50	5.3	25	39	4.7	60

Figure 11. Results with FGMRES(ls,10) and GMRESR(50,l_t,10), β=1 and h=1/100.

6 Recent results

In this section we give some recent results, which are subject to further study. First, we give some experiences with the LSQR switch for an indefinite system of equations. Then we report some experiments with GMRESR, in which we use a single-precision innerloop.

The LSQR switch. First we give a motivation of the LSQR switch. Thereafter we give a problem where the convergence of GMRESR is much faster using the LSQR switch.

We use the LSQR switch in the innerloop in the case that GMRES (nearly) stagnates. Due to the optimality property of GMRES it is easily seen that every other Krylov subspace method based on $K_k(A, r_0) = span\{r_0, Ar_0, \ldots, A^{k-1}r_0\}$ stagnates or breaks down. However, it is possible that LSQR, which is based on $K_k(A^TA, A^Tr_0)$, converges reasonably fast. Examples of such problems are given in [11] and [12]. The idea is that GMRESR with LSQR switch not only works if GMRES has a good convergence behavior but also if GMRES stagnates and LSQR converges reasonably fast. Furthermore if GMRES stagnates after some iterations, it is not necessary to restart with another iterative method; it is sufficient to change the iterative method in the innerloop (for instance LSQR).

In [2] we propose to relax the LSQR switch condition, instead of switching when $\|Au_{k,m}^{(0)} - r_k\|_2 = \|r_k\|_2$, we switch when $\|Au_{k,m}^{(0)} - r_k\|_2 \geq s\|r_k\|_2$, $s \leq 1$ for some suitable s close to 1. Below we describe some other GMRESR variants using a relaxed switch condition. We compare them for a discretization of the following partial differential equation:

$$-(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) + \beta(\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}) - 100u = f \text{ on } \Omega$$

$$u|_{\partial\Omega} = 0$$

where Ω is the unit square. The discretization is the same as the one used for the similar test problem given in Section 2. The linear system can be indefinite due to the final term on the left hand side of the PDE. We take $\beta = 10$, $h = 1/10$ and use the *trunclast* variant with $l_t = 10$.

We consider the following GMRESR variants:

GMRESR(5): innerloop consists of GMRES(5), combined with the strict LSQR switch ($s = 1$),

GMRESR1(5): innerloop consists of GMRES(5) followed by one LSQR iteration,

GMRESR2(5): innerloop consists of one LSQR iteration followed by GMRES(5),

GMRESR3(5): innerloop consists of GMRES(5), combined with a relaxed LSQR switch ($s = 0.99$),

GMRESR4(5): innerloop consists of GMRES(5), if $\|AU_{k,5}^{(0)} - r_k\| \geq 0.99\|r_k\|_2$ then GMRES(5) is followed by one LSQR iteration.

GMRESR(5) with $s = 1$ does not converge within 1000 iterations. The results using the other variants are given in Figure 12. Note that GMRESR1 and GMRESR3 have a reasonable good convergence behavior. This motivates us to combine the ideas between both, which leads to the GMRESR4 variant. The advantages of GMRESR4 are: it only uses an LSQR iteration if it is necessary, and the GMRES(5) results are not thrown away, as is done in the GMRESR3 variant. Furthermore it appears from Figure 12 that GMRESR4 has the best convergence behavior. All variants use approximately the same CPU-time per iteration.

The single-precision innerloop. In the GMRESR method, the innerloop calculates an approximate solution of $Au_{k,m}^{(0)} = r_k$. This approximation is used as search direction, so its accuracy only influences the convergence

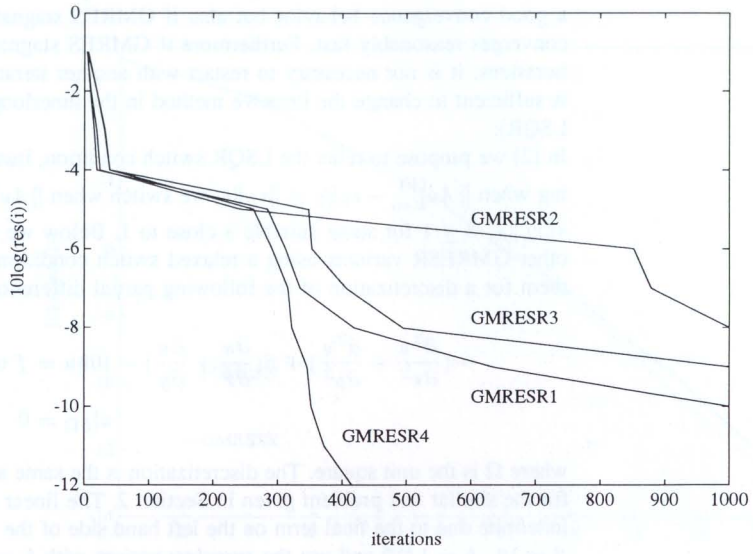


Figure 12. The convergence behavior of GMRESR variants.

but not the accuracy of the final solution vector. Since some computers calculate faster in single precision than in double precision, we have done experiments with a single-precision innerloop. The vector $c_k^{(0)} = Au_k^{(0)}$ should be calculated in double-precision. In such a case m_{opt} with respect to work can be chosen sligthly larger because the innerloop is cheaper. A comparable approach is given in [13] and [14]. In [14] they use as innerloop GMRES(m) in single precision (32-bits) and as outerloop an iterative refinement algorithm in double precision (64-bits).

The results for the testproblem of Section 2 are given in Figure 13. The CPU time of GMRESR(15) with single-precision innerloop is indeed less than the CPU time of GMRESR(10), whereas the final solution vectors have the same accuracy. With respect to memory one needs an extra single-precision copy of the matrix, however this increase is in general less than the decrease in memory caused by the fact that m can be chosen larger (and thus closer to m_{opt} with respect to memory) and the auxiliary vectors used in the innerloop are single precision.

Figure 13. Results for β=1,h=1/100 on the Convex C3820.

Method	Iterations	CPU time (s)
GMRESR(10)	36	4.4
GMRESR(15) single precision	27	2.9

We have also done experiments on one processor of a Cray Y-MP4/464. On this machine single-precision arithmetic (64 bits) is much faster than double-precision arithmetic (128 bits) which cannot be vectorized. Note that there are practical problems, where the system of equations is very ill conditioned. These problems can only be solved using a high accuracy iterative method. In these experiments we choose as termination criterion: $\|r_k\|_2/\|r_0\|_2 \leq 10^{-20}$. Note firstly that this test problem is only meant as an illustration for the use of a single-precision innerloop, secondly that it is not easy to obtain this high precision on a 64-bits computer. It follows from Figure 14 that GMRESR with a single-precision innerloop is much faster than with a double-precision innerloop. The solution vectors have the same accuracy.

In order to compare the machines we also did the same experiment as on the Convex. GMRESR(10) on the Convex is equal to GMRESR(10) with single-precision inner and outer loop on the Cray. It follows from Figure 14 that the Cray (1 processor) is 5.5 times faster for this problem.

Method	Innerloop	Outerloop	Iterations	CPU time (s)	Accuracy
GMRESR(20)	double precision	double precision	30	108	10^{-20}
GMRESR(20)	single precision	double precision	30	10.8	10^{-20}
GMRESR(10)	single precision	single precision	36	0.81	10^{-12}

Figure 14. Results for β=1,h=1/100 on the Cray Y-MP4/464.

On the Convex C3820 we have also applied GMRESR with single-precision innerloop on the discretized Navier-Stokes equations (see Section 2 and [6]). In these experiments GMRESR with single-precision innerloop is 25% faster than GMRESR, and the Bi-CGSTAB method.

7 Conclusions

We consider the GMRESR(m) method [2], which can be used for the iterative solution of a linear system $Ax = b$ with an unsymmetric and nonsingular matrix A.

Optimal choices for the parameter m are easily obtained and do not change very much for different problems. In most experiments we observe for GMRESR(m) a considerable improvement, in computing and memory requirements, in comparison with more familiar GMRES variants. Furthermore, it appears that in many experiments GMRESR(m) is a robust method even without activating the relaxed LSQR switch.

With respect to CPU time full GMRESR(m) seems to be the best variant. However, memory requirements can be so large that restarted and/or truncated GMRESR(m) should be used. From our experiments it appears that the *minalfa* truncation variant is the best strategy, which leads to a large decrease of memory requirements and only a small increase of CPU time.

We compare GMRESR(m) with GMRES, Bi-CGSTAB, and QMR. It appears that two easy to measure parameters: f and m_g, which depend on the used computer and on the properties of the system of equations

(dimension of the matrix, convergence behavior, sparseness, etc.), can be used to facilitate the choice of an iterative method.
In [5] a new GMRES-like method is proposed: FGMRES. It appears that full GMRESR is compatible with full FGMRES, however FGMRES can break down, and can only be restarted. From examples it follows that truncated GMRESR can be much better than restarted FGMRES.
We give some new results with respect to the relaxed LSQR switch. The best innerloop strategy seems to be: always apply GMRES(m), and if necessary do one LSQR iteration.
Finally, if one uses a computer on which single-precision arithmetic is faster than double precision arithmetic, and the condition number of A is not too large, then a single-precision innerloop saves CPU time.

Acknowledgement
I would like to thank H.A. van der Vorst for stimulating discussions and the suggestions for the GMRESR1 and GMRESR2 variants given in Section 6. This work was sponsored by the Stichting Nationale Computerfaciliteiten (National Computing Facilities Foundation, NCF) for the use of supercomputer facilities, with financial support from the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (Netherlands Organization for Scientific Research, NWO).

References

1
Saad, Y. and M.H. Schultz, *GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput. **7**, 856–869, 1986.

2
Vorst, H.A. van der and C. Vuik, *GMRESR: a family of nested GMRES methods*, Report 91-80, Faculty of Technical Mathematics and Informatics, Delft University of Technology, 1991, J. Num. Lin. Alg. Appl., to appear.

3
Vorst, H.A. van der, *Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of non symmetric linear systems*, SIAM J. Sci. Statist. Comput. **13**, 631–644, 1992.

4
Freund, R.W. and N.M. Nachtigal, *QMR: a quasi-minimal residual method for non-Hermitian linear systems*, Num. Math. **60**, 315–339, 1991.

5
Saad, Y., *A flexible Inner-Outer preconditioned GMRES algorithm*, SIAM J. Sci. Statist. Comput. **14**, 461–469, 1993.

6
Vuik, C., *Solution of the discretized incompressible Navier-Stokes equations with the GMRES method*, Int. J. Num. Meth. Fluids **16**, 507–523, 1993.

7
Paige, C.C. and M.A. Saunders, *LSQR: an algorithm for sparse linear equations and sparse least squares*, ACM Trans. Math. Soft. **8**, 43–71 1982.

8
Axelsson, O. and G. Lindskog, *On the eigenvalue distribution of a class of preconditioning methods*, Numer. Math. **48**, 479–498, 1986.

9
Vorst, H.A. van der and C. Vuik, *The rate of convergence of the GMRES method*, Preprint 654, University of Utrecht, Department of Mathematics, 1991, J. Comp. Appl. Math., to appear.

10
Jackson, C.P. and P.C. Robinson, *A numerical study of various algorithms related to the preconditioned conjugate gradient method*, Int. J. Num. Meth. Engng. **21**, 1315–1338, 1985.

11
Brown, P.N. *A theoretical comparison of the Arnoldi and GMRES algorithms*, SIAM J. Sci. Statist. Comput. **13**, 58–78, 1991.

12
Nachtigal, N.M., S.C. Reddy and L.N. Trefethen, *How fast are non symmetric matrix iterations*, SIAM. J. Sci. Statist. Comput. **13**, 778–795, 1992.

13
Zubair, M., S.N. Gupta and C.E. Grosch, *A variable precision approach to speedup iterative schemes on fine grained parallel machines*, Parallel Comp. **18**, 1223–1232, 1992.

14
Turner, K. and H.F. Walker, *Efficient high accuracy solutions with GMRES(m)*, SIAM J. Sci. Statist. Comput. **13**, 815–825, 1992.

SUPERCOMPUTER

Contributions

Parallel synchronous and asynchronous iterative methods to solve Markov chain problems

Abderezak Touzene, Brigitte Plateau

LGI-IMAG-Groupe Calcul parallèle, 46 avenue Félix Viallet, F-38031 Grenoble, France

©

Supercomputer **55**, X–3
Received July 1993

The aim of this work is to present well-suited parallel iterative methods for solving Markov chain problems on distributed memory machines. The main feature of such methods is reduced synchronization constraints between processors. We call these methods, asynchronous methods. Some experiments are given for comparisons with their synchronous counterparts.

1 Introduction

Systems to be studied are becoming more and more complex and hence their models are very large. To solve these models on a computer, we are faced with two constraints: speed and memory requirements. Parallel computation offers a good opportunity to overcome such constraints. The speed of computation depends on the number of processors working together on the problem, as well as the way the work is shared among the processors; reducing communication and synchronization between them. This last aspect is referred to as data and computation mapping. The memory requirement is solved by the use of distributed memory machines. If p denotes the number of processors of the machine, as each processor has its own local memory, the total memory available is multiplied by a factor p.

Let us now take a look at the problem that we want to solve. Typically, we are interested in finding the stationary probability vector π of a Markov chain with T states such as $\pi = \pi P$, where P is a probability transition matrix associated with the Markov chain. When T is large, iterative methods are more attractive than direct methods regarding the computational cost.

We consider a standard iterative method, illustrated by:

$$\pi^{(t+1)} = \pi^{(t)} P \qquad (1)$$

where $\pi^{(t)}$ is the vector computed at iteration number t. In this paper we use Jacobi and Gauss-Seidel-like iterative schemes defined as follow:

– Jacobi-like iterative scheme

$$\pi_i^{(t+1)} = f_i(\pi_1^{(t)}, \pi_2^{(t)}, \ldots, \pi_T^{(t)}), \forall i \in [1..T].$$

– Gauss-Seidel like iterative scheme

$$\pi_i^{(t+1)} = f_i(\pi_1^{(t+1)}, \ldots, \pi_{i-1}^{(t+1)}, \pi_i^{(t)}, \ldots, \pi_T^{(t)}), \forall i \in [1..T].$$

where $f_i, \forall i \in [1..T]$ can be any linear operators.

The parallel implementation of (1) is in fact based on the parallel vector-matrix multiplication. We use the block column decomposition method [1] because it supports all the iterative schemes to be presented. At any iteration, two steps are performed:

1. first, each processor updates a set of vector components of π according to the mapping of columns of the matrix;
2. a second step consists of synchronization and data exchange among processors to ensure the progression of the iterative process (1).

Because of the strong synchronization at the second step (each processor cannot begin the computation of iteration $(t+1)$ until it receives all the data of the iteration (t), this iterative process is called a *synchronous* method.

Chaotic relaxation, an *asynchronous* iteration principle, has been introduced by Chazan and Miranker [2], to solve linear systems. In the asynchronous implementation of the iteration $\pi^{(t+1)} = f(\pi^{(t)})$, the processors do not need to wait for the updates generated at the previous iteration; each processor modifies its vector component regardless of the others. If the current value of a component, updated by another processor, is unavailable, then an outdated value of this component is used. Furthermore, the processors are not obliged to communicate their results after each iteration, but only when desired.

The asynchronous computation offers the following advantages:

– reduction of the communication time and the synchronization delay;
– the improvement of convergence due to the Gauss-Seidel effect.

A theorical convergence study of asynchronous iteration is given in [1, 3–5], but only a few parallel implementations have been studied [5, 6] and these experiments have been performed on shared memory machines.

The aim of this paper is to present the parallel implementation of asynchronous iterative schemes on distributed memory machines.

In section 2, we present a Jacobi-like basic synchronous iterative method. This method is described in order to compare the different asynchronous schemes that we develop in this paper. One aspect of asynchronous computation is the integration of the last updates of each vector component. This idea exists also in Gauss-Seidel iteration schemes. Applying a strict Gauss-Seidel-like iteration scheme leads to a total ordering among tasks updating each vector component. We propose to apply this partially and eliminate some costly precedence constraints. This idea is developed in section 3.

Another aspect of asynchronous computation is the reduction of communication among processors. We study the delay introduced in the methods in sections 4 and 5. This delay notion stems from the fact that, in the updates at any iteration number (t), we use some values of vector components generated at iteration $(t-k)$, where k is a positive inte-

ger measuring the "delay". In asynchronous computation, which favors a Gauss-Seidel effect, we use parallelism between communication and computation on each processor if it is possible; as in transputer-based machines. In this case, we propose an implementation very close to the asynchronous definition presented in [1]: each processor executes two kinds of processes. One process computes the new values of vector components using the more recent updates where the second is devoted to broadcast continuously the new values generated at each processor. This method is presented in section 6. Section 7 consists of an experiment with the different iterative schemes. A comparison between these schemes and the basic synchronous iterative method is given. Finally, we give some conclusions in section 8.

2 The basic synchronous iterative scheme

We are interested in finding the stationary probability vector π of a Markov chain with a transition probability matrix P using the iteration $\pi^{(t+1)} = \pi^{(t)} P$. This iterative method is known as *the power method* (a Jacobi-like method) [7, 8]. First we choose any starting vector $\pi^{(0)}$ and then proceed by successive iterations until a given convergence criterium is satisfied. From the parallel point of view, we assume that T, the size of the matrix P, is divisible by p, the number of processors. To ensure a balanced computation cost among processors, each processor must update $\frac{T}{p}$ consecutive components of vector π. To do this, each processor numbered p_l, $l = 0, \ldots, p-1$, responsible for updating the consecutive vector components numbered from $l\frac{T}{p} + 1$ to $l\frac{T}{p} + \frac{T}{p}$, must have in its local memory an appropriate column block of P. This column block consists of consecutive columns of the matrix P. Typically, at any iteration (t), each processor processes a vector-matrix product of size $\frac{T}{p}$ (the *Product()* process). This method is called column block decomposition [1, 9]. Notice that the entire vector π is copied on each processor. After the updates by each processor, and in order to form the vector $\pi^{(t+1)}$ on each processor, a communication step is performed. This communication step is the *AllToAll()* procedure which is characterized by: *each processor sends its updated components to all other processors.*

The cost of this communication step depends on the network topology and on the communication procedure. In [10, 11] optimal algorithms are given respectively for hypercube and mesh networks.

The implementation of the synchronous basic iterative scheme is given by the following algorithm:

Algorithm 0.
Begin
t = 1
While *NotConvergence*
Product();
AllToAll();
ConvergenceTest();
t = t+1;
Endwhile
End

This basic synchronous method, allows us to compare with the asynchronous methods to be presented in the remainder of this paper.

3 The iterative scheme using local Gauss-Seidel effect

It has been shown [8] that the use of the more recent updates of the vector components seems to improve the convergence speed of any iterative scheme; this is the case in the Gauss-Seidel-like iterative schemes. The problem of the Gauss-Seidel scheme lies in the precedence constraints when updating the vector components (first update the component π_i and then π_{i+1} and so on). However, Jacobi-like iterative schemes have no precedence constraint when updating the vector components and are therefore well adapted to parallel computation. We propose a *mixed* method, that uses the parallelism of Jacobi-like methods and a local updating scheme according to Gauss-Seidel. In other words:

- Do the same distribution of data on processors as in the Jacobi-like scheme (see section 2).
- Apply a Gauss-Seidel updating scheme on the local components at each processor. If we consider the updating on the processor p_l and J the set of vector components to be updated by this processor, we give the following scheme:

$$\pi_i^{(t+1)}, i \in J \text{ is computed from } \begin{cases} \pi_j^{(t+1)} & l(\frac{T}{p}) + 2 \leq j < i \\ \pi_j^{(t)} & \text{elsewhere} \end{cases} \quad (2)$$

The algorithm of this method is:
Begin
t = 1
While *NotConvergence*
ProductGaussEffect() {* Product using the formula (2) *}
AllToAll();
ConvergenceTest();
t = t+1;
Endwhile
End

This algorithm is motivated by a possible improvement in convergence.

4 The iterative scheme using the delay k

In this scheme, we want to exploit a delay to reduce the communication between processors (the *AllToAll()* process). The delay expresses the fact that in the updates at iteration number t, we use outdated values of vector components, generated at iteration $(t-k)$, where k is a positive integer. The idea is to perform k iterations without communicating the results to the other processors. After this k isolated iterations, each processor broadcasts its most recent updates to all processors:

Algorithm 1.
Begin
t = 1
While *NotConvergence*
If *t* Mod *k+1 = 0*
{∗ do the communication step after each k iterations ∗}
Then *AllToAll();*
ConvergenceTest();
Else *Product();* {∗ or *ProductGaussEffet()* ∗}
t = t+1;
Endwhile
End

We have seen previously that the convergence speed seems to be related to the use of the more recent updates (Gauss-Seidel effect). The delay method goes in the opposite direction of this principle. Thus we expect that the convergence of this method is worse than in the case of the basic iterative scheme. A compromise should be possible between the speed of convergence and the reduction of communication.

It is clear that this scheme reduces the communication because of the isolation of each process during k iterations.

The question is: can we also reduce the computation time? This is studied in the next section.

5 Computational cost amelioration in the delay k scheme

When each processor iterates on its local vector components without sending or receiving new values of components updated by other processors, we see an invariant quantity in the updating expression of these components during the k iterations in isolation. We propose an optimized method when computing the vector matrix product on each processor. The idea is to compute these constant quantities only once during the k iterations.

Let J be the set of index vector components of π to be updated by any processor. The cardinality of this set is given by:

$$\mathrm{Card}(J) = \frac{T}{p}.$$

Knowing the vector $\pi^{(t)}$, we want to perform k local iterations. We compute the vector $\pi^{(t+1)}$ using the *Product()* process:

$$\pi^{(t+1)}_{(j\in J)} = \sum_{l\notin J} \pi^{(t)}_l P_{lj} + \sum_{l\in J} \pi^{(t)}_l P_{lj}. \tag{3}$$

And then the vector $\pi^{(t+2)}$:

$$\pi^{(t+2)}_{(j\in J)} = \sum_{l\notin J} \pi^{(t)}_l P_{lj} + \sum_{l\in J} \pi^{(t+1)}_l P_{lj}. \tag{4}$$

Notice that the quantity $\sum_{l\notin J} \pi^{(t)}_l P_{lj}$ is invariant for all iteration numbers $m \leq t+k$; we shall show how to evaluate it. From expression (3) and (4) we yield

$$\pi^{(t+2)}_{(j\in J)} = \pi^{(t+1)}_{(j\in J)} - \sum_{l\in J} \pi^{(t)}_l P_{lj} + \sum_{l\in J} \pi^{(t+1)}_l P_{lj}.$$

For all iteration numbers $m \leq t+k$, the vector $\pi^{(m)}$ is computed by the following formula:

$$\pi^{(m)}_{(j\in J)} = \pi^{(m-1)}_{(j\in J)} - \sum_{l\in J} \pi^{(m-2)}_l P_{lj} + \sum_{l\in J} \pi^{(m-1)}_l P_{lj}. \tag{5}$$

The computational cost using formula (3) is $O(\frac{T^2}{p})$ operations (additions and multiplications). By using formula (5), the cost is $O(\frac{T^2}{p^2})$ operations, a considerable gain when the number of processors p is large.

Algorithm 1 optimized.
Begin
Product();
AllToAll;
Product();
t = 1
While *NotConvergence*
If *t* Mod *k = 0*
Then *AllToAll();*
Product();
ConvergenceTest();
Else *ProductOptimized();* {∗ see the formula (5) ∗}
t = t+1;
Endwhile
End

In the following section, we examine the asynchronous iterative scheme which favors the Gauss-Seidel effect.

6 Asynchronous free scheme

In the asynchronous scheme, the processors use, in their updates, incoming new values as soon as they are received from the other processors.

When the target parallel machine allows parallelism between communication and computation, the idea is to execute in parallel the computation process (updates on the vector components) and the communication process (to send the new component values). More precisely, the communication process is a loop of the *AllToAll* procedure. The motivation of this scheme is to favors the Gauss-Seidel effect. In practice, this will be the case if the computation cost is larger than the time taken to achieve one total exchange of data.

Algorithm 2.
Begin
ProcRun(ProductLooped())
{∗ Concurrent execution of the product process ∗}
While *NotConvergence*
AllToAll();
ConvergenceTest();
Endwhile
End

The *ProductLooped()* process is in fact a loop on *Product()*. This loop is controlled by the convergence criteria. When the execution time of the process *Product()* exceeds the time to achieve the *AllToAll()* process, many calls of *AllToAll()* happen during one call to *Product()*. This results in increased communication cost compared to the synchronous version. Theoretically, the cost of added communication is small because of the assumption of parallelism between communication and computation. But, in practice this cost depends strongly upon the overlapping ratio between communication and computation in parallel machines.

In the following section, we carry out experiments of all the presented iterative schemes on the distributed memory machine MEGANODE [12] configured as a wrap-around mesh of (11×11) transputers.

7 Experimental results

To test different iterative schemes, we first use the matrix type frequently arising in Markovian modelling. Generally these matrices are sparse.

Next, we test with another frequently used matrix type. These matrices are associated with Nearly Completely Decomposable models (NCD). This type of matrices are block structured. We continue our experimentation with tridiagonal block matrices (Quasi Birth and Death models). Finally, we test a real model. This model represents a random walk on a 2-D grid.

All these matrices are all irreducible, in order to fulfil the conditions of convergence theorems for asynchronous computation [1, 3–5]. They have a loop on all states of their associated Markov chain to ensure their aperiodicity and satisfy a Mitra condition [5] for stochastic matrices: there exists at least an index i such $P_{ii} > 0$.

For all matrix types, we observe the mean number of iterations and their corresponding computation time of randomized matrices.

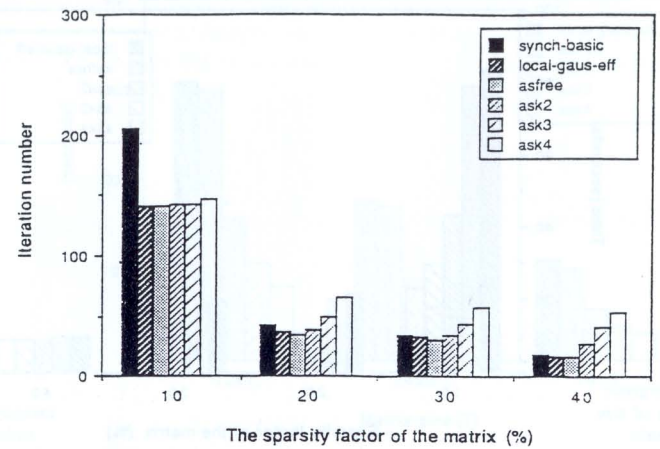


Figure 1. Number of iterations for matrices of size 3267 (sparse matrix type).

Sparse matrices

We now study the number of iterations and their corresponding computation time while varying the matrix sparsity.

Figure 1 shows that experiments confirm that the number of iterations of delay k schemes increases with k. We also see the superiority of local Gauss-Seidel scheme over the basic iterative method (power method). We notice the small gain of iteration brought by the free asynchronous scheme. This is the case for all matrix types. Probably this is due to the ratio of computation and communication speed of our target machine. What is surprising is the significant saving in execution time when using the k delay schemes with the optimized version. Figures 2, 3 and 4 show that this schemes may save about half of the execution time.

Nearly decomposable (NCD) type matrices

The same experiment is performed with diagonal block matrices, when the off-block diagonal elements of the matrices are less then $\epsilon = 10^{-5}$. Figure 2 shows the gain using k delay scheme (about $\frac{1}{3}$) when the size of the diagonal block coincides with the cardinal number of the set of indices to be updated by the processors. In this case, this all happens as if each processor is dedicated to solve an independent subsystem because of the weak interactions between subsystems (the decomposability factor, is small).

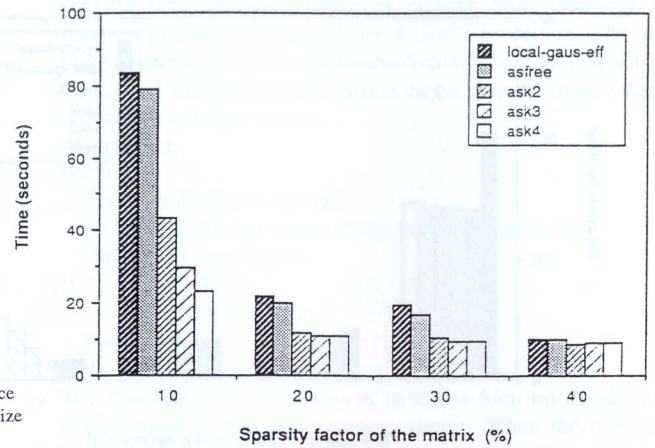


Figure 2. Convergence time of matrices of size 3267 (sparse matrix type).

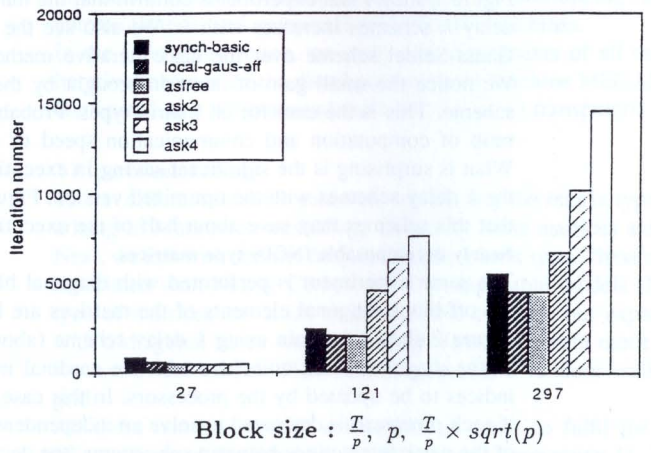


Figure 3. Convergence time for the NCD case matrix with block size $\frac{T}{p}$.

Tridiagonal block matrix type

For tridiagonal block matrices, the same phenomenon as in the section above occurs; this is shown by Figures 5 and 6.

Random walk on 2-D grid

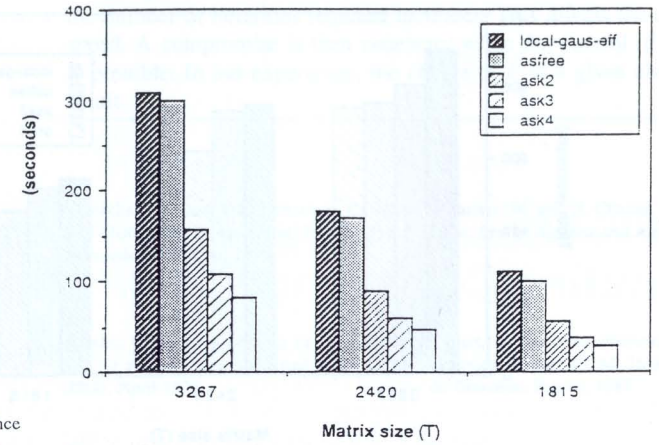


Figure 4. Convergence time for the random walk on 2-D grid.

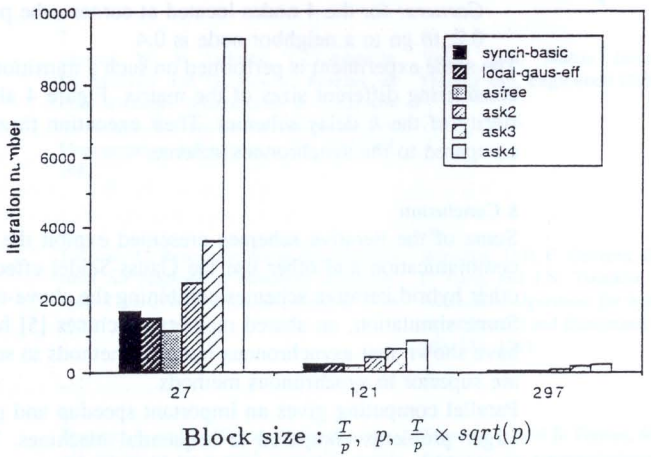


Figure 5. Number of iterations for matrices of size 3267 (tridiagonal matrix type).

Consider a $(\frac{T}{p} \times p)$ sized grid. Each grid node represents a given state of Markov chain (probability to find the walker at any grid node). The transition probability from any node to another is given as follows.

– *Inner nodes:* the probability that the walker stays at the same node is 0.2 and the probability to go to one of the four neighbor nodes is

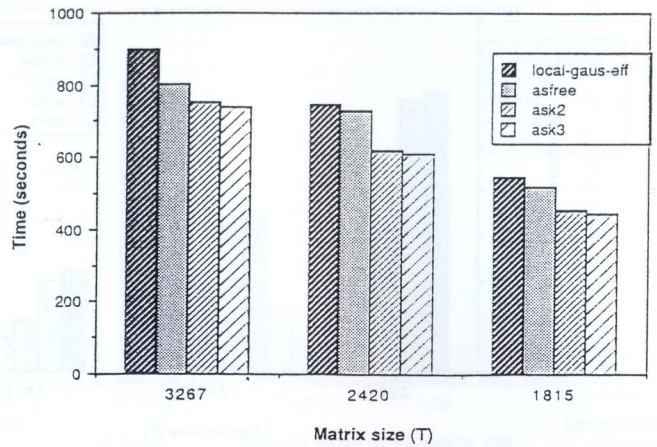


Figure 6. Convergence time for tridiagonal block matrix case with the blocs size is $\frac{T}{P}$.

0.2.

- *Boundary nodes (exept corners:* the probability to stay is at the same node is 0.1, the probability to go to a neighbor node is 0.3.
- *Corners:* for the 4 nodes located at corners, the probability to stay is 0.2, to go to a neighbor node is 0.4.

The same experiment is performed on such a transition probability matrix considering different sizes of the matrix. Figure 4 also shows the superiority of the k delay schemes. Their execution times are about halved compared to the synchronous scheme.

8 Conclusion

Some of the iterative schemes presented exploit the delay for reducing communication and other use the Gauss-Seidel effect. We can imagine other hybrid iterative schemes combining the above-mentioned schemes. Some simulation, on shared memory machines [5] have been done and have shown that asynchronous iterative methods to solve Markov chains are superior to synchronous methods.

Parallel computing gives an important speedup and permits one to treat larger problems compared to sequential machines. This paper considered parallel methods to solve eigenvector problems applied to Markov chain modelling. We have proposed some iterative asynchronous schemes and we have compared them to the *power method*, a basic synchronous scheme. The results show that the free asynchronous scheme leads to a weak improvement by the Gauss-Seidel effect, and the control of this scheme depends principally on the computing/communication ratio. On the other hand, the k delay schemes give good results. The gain of these schemes depends on delay k. This delay must be kept bounded to ensure convergence (speed). The experiments show that when k increases, the cost of synchronization and communication decreases. But unfortunately, the number of iterations required increases. This affects the convergence speed. A compromise is then necessary when the control of the delay k is possible. In our experience, the choice of $k = 3$ gives always a good result.

References

1
Tsitsiklis, J.N. and D.P. Bertsekas, *Parallel and Distributed Computation*, Prentice-Hall, International Edition, 1989.

2
Miranker, W. and D. Chazan, *Chaotic relaxation, Linear Algebra and Appl.* **2**, 1969.

3
Baudet, G.M., *Asynchronous iterative methods for multiprocessors*, Journal of the ACM **25(2)**, April 1978.

4
Robert, F., *Iterations discretes asynchrones*, Technical Report 671M, IMAG, Universite de Grenoble, France, 1987.

5
Lubachevsky, B. and D. Mitra, *A chaotic asynchrounous algorithm for computing the fixed point of nonnegative matrix of unit spectral radius*, Journal of the ACM **33(1)**, Jan 1986.

6
LeGall, F. and J. Bernussou, *About some iterative synchronous and asynchronous methods for Markov chain distribution computation*, IFAC - Munich, 1987.

7
Stewart, W., E. Gelenbe, J. Labetoulle, M. Metivier and G. Pujolle, *Réseaux de Files d'Attente, Modélisation et Traitement Numérique*, Ed. des hommes et techniques, Monographies informatiques de l'AFCET, 1981.

8
Varga, R.S., *Matrix Iterative Analysis*, Prentice-hall, Englewood Cliffs NJ, 1963.

9
Touzene, A., *Resolution des modèles Markoviens sur machines à mémoires ditribuées*, PhD Thesis, INP Grenoble, France, Sept 1992.

10
Bertsekas, D.P., C. Ozveren, G.D. Stamoulis, P. Tseng and J.N. Tsitsiklis, *Optimal communication algorithms for hypercubes*, Journal of Parallel and Distributed Computing **11**, 263–275, 1991.

11
Touzene, A. and B. Plateau, *Optimal multinode broadcast on a mesh connected graph with reduced bufferization*, in: "Distributed Memory Computing", 2nd European Conference, EDMCC2, Munich, FRG, 1991.

12
Touzene, A. and B. Plateau, *Mesures de performance des communications du meganode à 128 transputers*, La lettre du transputer et des calculateur distribués **7**, September 1990.

The 1994 Scalable High Performance Computing Conference – SHPCC94

The 1994 Scalable High Performance Computing Conference (SHPCC94) is a continuation of the highly successful Hypercube Concurrent Computers and Applications (HCCA), and Distributed Memory Concurrent Computing (DMCC) conference series. SHPCC takes place biennially, alternating with the SIAM Conference on Parallel Processing for Scientific Computing.

The invited speakers include:

Guy Blelloch (Carnegie Mellon University);
Phil Colella (University of California, Berkeley);
David Culler (University of California, Berkeley);
Monica Lam (Stanford University);
Marc Snir (IBM T.J. Watson Research Center).

SHPCC94 will provide a forum in which researchers in the field of high-performance computing from government, academia, and industry can presents results and exchange ideas and information. SHPCC94 will cover a broad range of topics relevant to the field of high-performance computing. These topics will include, but are not limited to, the following: Architectures, Load Balancing, Artificial Intelligence, Linear Algebra, Compilers, Neural Networks, Concurrent Languages, Non-numerical Algorithms, Fault Tolerance, Operating Systems, Image Processing, Programming Environments, Large-scale Applications, Scalable Libraries, C^{++}.

The SHPCC94 program will include invited talks, contributed talks, posters, and tutorials. SHPCC94 will take place at the Holiday Inn Convention Center in Knoxville, Tennessee, USA.

Poster presentations are intended to provide a more informal forum in which to present work-in-progress, updates to previously published work, and contributions not suited for oral presentation. Poster presentations will not appear in the Conference Proceedings.

Half-day and full-day tutorials provide opportunity for a researchers and students to expand their knowledge in specific areas of high-performance computing.

For further information contact

David W. Walker
Oak Ridge National Laboratory
Bldg. 6012/MS-6367
P. O. Box 2008,
Oak Ridge, TN 37831-6367, USA
fax: +1 615 5740680
e-mail: walker@msr.epm.ornl.gov.

SUBSCRIPTION FORM

I want to subscribe for **SUPER**COMPUTER for 1993 for 255 Dfl or $155.

....
date signature

Send **SUPER**COMPUTER to the following address:

....
first name surname

.....
institute/company

.....
address

.....
VAT nr.

.....

Send the invoice to ❒ the address above ❒ the following address:

....
first name surname

.....
institute/company

.....
address

ASFRA, Voorhaven 33, 1135 BL Edam, The Netherlands