Deteriorating Convergence for Asynchronous Methods on Linear Least Squares Problems*

Trond Steihaug and Yasemin Yalçınkaya

University of Bergen,
Department of Informatics,
Bergen, Norway

Abstract. A block iterative method is used for solving linear least squares problems. The subproblems are solved asynchronously on a distributed memory multiprocessor. It is observed that an increased number of processors results in deteriorating rate of convergence. This deteriorating convergence is illustrated by numerical experiments. The deterioration of the convergence can be explained by contamination of the residual. Our purpose is to show that the residual is contaminated by *old information*. The issues investigated here are the effect of the number of processors, the role of *essential neighbors*, and synchronization. The characterization of old information remains an open problem.

1 Introduction

In this paper, a block iterative method is used for solving sparse linear least squares problems. A general framework for this method is introduced by Dennis and Steihaug [4], and the preliminary tests in [4] indicate that this method leads quickly to cheap solutions of limited accuracy.

Due to the rapid development and increasing usage of parallel computers and distributed computing, it is now important to adapt the methods to the new architectures. Baudet's [1] experimental results on systems of linear equations show a considerable advantage for iterative methods on parallel computers with no synchronization. This leads to experimentation with *totally asynchronous* [2] block iterative methods for the solution of linear least squares problems.

The block iterative method [4] partitions the columns of the coefficient matrix into disjoint blocks of columns and then projects the updated residual into each column subspace. This algorithm, which is called *column oriented successive subspace correction* (CSSC) in [8], is, in fact, Gauss-Seidel iteration on the normal equations. Each subproblem is substantially smaller than the original problem and hence is solved directly using QR factorization [7] and semi-normal equations (SNE) [3] on one processor. Each processor computes a correction of the solution vector restricted to the variables associated with the blocks of columns. The computation requires the residual, which is the global data. Each processor uses the residual available at the start of the computations without waiting for

* This research is supported by The Research Council of Norway.

the newest data. This way, the disadvantages resulting from the execution of synchronization primitives are avoided.

We do not address the issues of timing and speedup in this paper. Some timing and speedup results can be found in [11].

In the following, we first give the framework of the block iterative method for the linear least squares problem and state the sequential algorithm. The subproblems in this algorithm are to be solved using a direct method. A discussion of suitable factorization techniques for the subproblems is found in [10]. Then, we introduce the totally asynchronous algorithmic model and the algorithm for the asynchronous implementation of the method. In Section 4, the results of our experiments are presented. In the asynchronous implementation it is observed that increased number of processors results in contaminated residual and hence deteriorating convergence. This is due to the existence of *old information* in the system.

2 The Linear Least Squares Problem

Let A be an m by n real matrix, $m \geq n, b \in \mathbb{R}^m$. Let M be an m by m positive definite matrix. The weighted linear least squares problem is:

$$\min_{x \in \mathbb{R}^n} \| Ax - b \|_M, \tag{1}$$

where $\|y\|_M^2 = y^T M y$.

The columns of A are divided into g blocks $A_1, A_2, \ldots, A_g$, where $A_i \in \mathbb{R}^{m \times n_i}$. Assume, without loss of generality, that each A_i has full rank. The least squares problem (1) is equivalent to

$$\min\{\|A_1 x_1 + A_2 x_2 + \cdots + A_g x_g - b\|_M : x_i \in \mathbb{R}^{n_i}, i = 1, 2, \ldots, g\} \ . \tag{2}$$

Suppose that x^k is an approximation to a solution x^* to (1), and x^k is divided into $x_1^k, x_2^k, \ldots, x_g^k$ as above. Then from (2), the following successive replacements iteration can be obtained:

for $i = 1, 2, \ldots, g$ **do**
  `Solve for` $x_i^{k+1} \in \mathbb{R}^{n_i}$:
    $\min \| \sum_{j=1}^{i-1} A_j x_j^{k+1} + A_i x_i^{k+1} + \sum_{j=i+1}^{g} A_j x_j^k - b \|_M$.

This is equivalent [4, 12] to:

for $i = 1, 2, \ldots, g$ **do**
  $\hat{r} = r^{k+(i-1)/g}$.
  `Solve for` s_i^k : $\min\{\|A_i s_i + \hat{r}\|_M : s_i \in \mathbb{R}^{n_i}\}$. (3)
  `Update the residual:` $r^{k+i/g} = r^{k+(i-1)/g} + A_i s_i^k$.
  `Update the solution:` $x^{k+i/g} = x^{k+(i-1)/g} + \bar{s}_i^k$.

For $s_i \in \mathbb{R}^{n_i}$, introduce the vector $\bar{s}_i \in \mathbb{R}^n$, which is obtained by starting with a zero vector and placing the nonzero entries of s_i in the right positions.

This is block Gauss-Seidel iteration on the normal equations for (1). The residual $\hat{r} = r^k$, gives block Jacobi iteration on the normal equations. The intermediate residual is a combined Jacobi and Gauss-Seidel method. With the introduction of a relaxation parameter successive over-relaxation (SOR) method is obtained.

Algorithm 1.

Subdivide A into g blocks.
Choose $0 < \omega_i < 2, i = 1, 2, \ldots, g$.
Choose $x_i^0, i = 1, 2, \ldots, g$, $x^0 = \sum_{i=1}^{g} \bar{x}_i^0$.
Compute $r^0 = Ax^0 - b$.
for $k = 0$ **step** 1 **until** *convergence* **do**
  for $i = 1, 2, \ldots, g$ **do**
    $\hat{r} = r^{k+(i-1)/g}$.
    Solve for s_i^k : $\min\{\|A_i s_i^k + \hat{r}\|_M\}$.
    $r^{k+i/g} = r^{k+(i-1)/g} + \omega_i A_i s_i^k$.
    $x^{k+i/g} = x^{k+(i-1)/g} + \omega_i \bar{s}_i^k$.
  Check for *convergence.*

The series of approximations $\{x^k\}$ from Algorithm 1 converge to x^*, a solution of the least squares problem (1), and $||r^k||_M$ is strictly monotonically decreasing [4].

3 Parallelization

In this section, we consider the parallel implementation of Algorithm 1, and formulate the main algorithm used in the experiments. First, we will see how we can get the parallel version of the sequential algorithm at hand, and then we will point out some of the advantages and disadvantages of asynchronous computation over the synchronous mode.

Jacobi type of iterations are straightforward to implement in parallel. In Algorithm 1, if use $\hat{r} = r^k$, we get Jacobi method. The main computation in the inner loop is the solution of (3). This system can be solved concurrently for each block i on multiple processors provided that the submatrices A_i are available on the processors. The processors, after computing their corrections on block components of x have to synchronize at the end of the loop before starting with the next loop. Now that we are able to compute the corrections from each block in one step, we have gained a considerable advantage over the sequential algorithm in terms of parallelization. In synchronized algorithms, the faster processors waiting for the slower ones to complete their computations causes an overhead. To get higher utilization of the available CPU power we can remove synchronization and the restriction on the order of the updates. By removing synchronization from the synchronous Jacobi algorithm and letting $\hat{r}$

get the latest available value of the residual in the system, we obtain a totally asynchronous algorithm.

Asynchronous algorithms reduce the synchronization penalty caused by fast processors waiting for slow processors to complete the computations, and for slow communication channels to deliver messages. The reason is that processors can execute more updates when they are not constrained to wait for the results of the computation on other processors. The only requirement on the computation of the updates is that, eventually, the values of an early update cannot be used any more in further evaluations. This condition is met as long as no processor falls out of the system. However, removing synchronization brings out the danger that the updates are performed on the basis of outdated (old) information.

An important disadvantage of asynchronism is that it can impede convergence properties that the algorithm may possess when executed synchronously or sequentially. In some cases, it is necessary to place limitations on the size of communication delays to guarantee convergence. Necessary conditions for the convergence of linear problems is given in Bertsekas and Tsitsiklis [2].

Before giving the asynchronous implementation of Algorithm 1, we will introduce the totally asynchronous algorithmic model.

3.1 The Totally Asynchronous Algorithmic Model

Let $\mathcal{T} = \{1, 2, \ldots\}$ be a set of times at which one block x_i of x is updated by some processor, and $\mathcal{T}^i$ = set of times at which x_i is updated.

The processor computing s_i may not have access to the most recent values of x_j at the time of the update. For $t \in \mathcal{T}^i$, $s_i(t)$ is computed using a residual $\hat{r} = \sum_{j=1}^{g} A_j x_j(\tau_j^i) - b$, where $\tau_j^i(t)$ are times satisfying

$$0 \leq \tau_j^i(t) \leq t - 1 \ .$$

At all times $t \notin \mathcal{T}^i$, x_i is left unchanged and

$$x(t) = x(t-1) + \bar{s}_i(t), \ t \in \mathcal{T}^i \ .$$

3.2 Asynchronous Implementation

The next algorithm is the asynchronous implementation of Algorithm 1 on an MPMD machine with $p = g + 1$ processors. In the algorithm, processor p_0 is used as the master and processors p_i, $i = 1, 2, \ldots, g$ act as slaves. **Send** and **Receive** are communications with the master. **Broadcast** is done by the master processor.

Algorithm 2.

`Subdivide` A `into` g `blocks.`
`Choose` $0 < \omega_i < 2, i = 1, 2, \ldots, g.$
`Choose` $x_i(0), i = 1, 2, \ldots, g, \ x(0) = \sum_{i=1}^{g} \bar{x}_i(0).$
`Compute` $r(0) = Ax(0) - b.$

```
Initiate each processor i = 1, 2, ..., g :
    Receive(A_i, p_0).
    Receive(x̄_i(0), p_0).
    Receive(ω_i, p_0).
    Preprocess.
Broadcast(r(0)).
                        {Let t be a global counter of corrections and}
                    {let t_1^i and t_2^i be two consecutive elements in T^i}
t = 0.
while not termination do
    if slave then
        Solve for s_i :
            min{||A_i s_i + r(t_1^i)||_M : s_i ∈ R^{n_i}}.
        Update x_i : x_i = x_i + ω_i s_i.
        Send(A_i s_i, p_0).
        Receive(r(t_2^i), p_0).
    else                                                    {master}
        Receive(A_i s_i, p_i).              {s_i computed using r at t_1^i}
        r(t + 1) = r(t) + ω_i A_i s_i.
        Check for termination.
        if not termination then
          t = t + 1;
          Send(r(t), p_i).                                  {t_2^i = t}
```

4 Experiments

Test problems used in the experiments are taken from Harwell-Boeing sparse matrix test collection [5]. All the graphs refer to problem ASH958. Blocks are formed by taking blocks of consecutive columns. For the specified problem, the number of blocks g is 30. Each subproblem is solved using QR factorization and SNE. Both parallel and sequential implementations are done on Intel Paragon. Static assignment of blocks to processors is chosen to avoid the overhead of assignments during the computation phase. The number of processors p reported is the number of slave processors.

Algorithm 1 is a block Gauss-Seidel iteration which can be converted to Jacobi type iteration by taking $\hat{r} = r^k$, and the intermediate residual is a combined Jacobi and Gauss-Seidel iteration. To verify this, the residual is "fixed" for p-updates, i.e. $\hat{r} = r^{k+c/g}$ in (3), where $c = p\lfloor i/p \rfloor$. The quantity $\lfloor \cdot \rfloor$ is the largest integer not greater than its argument. Assume for simplicity that g mod $p = 0$. This routine implements Gauss-Seidel updates with a Jacobi iteration on groups of g/p block components. In this implementation $p = g$ gives the block Gauss-Seidel method, and $p = 1$ gives the block Jacobi method. We see in Fig. 1 that the resulting convergence is between sequential Gauss-Seidel and Jacobi methods acknowledging our statement. Increased number of processors results in increased deterioration of the convergence of the residual.

In the second experiment, a "time-lag" of p is used, where $\hat{r} = r^{k+(i-p)/g}$ in (3). The same block assignment as in the former case is used. Again, an increase in the number of processors results in increased deterioration as seen in Fig. 2. To state this result in a more formal way, let

$$\rho(p) = \sup_{x^0} \limsup_{k\to\infty} \|x^k - x^*\|^{1/k}$$

be the average rate of convergence using p processors. For a special class of linear systems Elsner, Neumann and Vemmer [6] prove $\rho(p+1) \geq \rho(p)$.

In the next experiment, a totally asynchronous iteration on different number of processors p, $p \leq g$, and $g \bmod p = 0$, is implemented. We see in Fig. 3 that for a small number of processors the rate of convergence lies in the neighborhood of sequential Gauss-Seidel method. If we increase the number of processors further to make better use of the available CPU power, the rate of convergence is degraded. In an asynchronous implementation, the order of the updates may change, and also the the total number of updates. In five different runs of the totally asynchronous implementation on $p = g$ processors, the number of updates before convergence varies between 958 and 1046.

An asynchronous implementation on homogeneous processors with negligible communication delays will, after some time, be almost cyclic in the updates. Let t_1 and t_2 be two consecutive updates of block i, $t_1, t_2 \in \mathcal{T}^i$. At t_2 the correction $s_i(t_1)$ is the solution of:

$$\min\{\|A_i s_i + r(t_1))\|_M\} \ .$$

The updated solution and residual are:

$$x(t_2) = x(t_2 - 1) + \omega_i \bar{s}_i(t_1), \quad r(t_2) = r(t_2 - 1) + \omega_i A_i s_i(t_1),$$

and for a cyclic implementation with cycle length g with p processors, an update will be computed with a residual which is p time units old. Hence, $t_1 = t_2 - p$, and we have a sequential implementation with time-lag of p.

We need to consider the effect of dependence between blocks on the convergence rate in the asynchronous implementation. Let $E_i = \{j \mid$ block i and block j have nonzero elements on the same row positions$\}$ be called the *essential neighbors* [9] of block i. Let $t_1, t_2 \in \mathcal{T}^i$ be consecutive times of update from block i. When we update the residual and approximate the solution at time t_2, $A_i^T r(t_2) = 0$ and $s_i(t_2) = 0$, unless any block $j, j \in E_i$ has sent an update between t_1 and t_2. In the next experiment, to avoid zero corrections, block i is forced to wait until an update from block $j, j \in E_i$ arrives at the master. We use $p = g$ processors in a heterogeneous environment with each processor's speed varying with a factor between 1 and 7. In Fig. 4, the curve marked WFE illustrates the implementation where the processors wait for their essential neighbors. The time between two nonzero updates of block i will be shorter if the processor waits for an essential neighbor of block i. Hence, we expect that the time-lag on the average will be reduced. In the numerical experiments, we see a decrease in the deteriorations, and as a result, an improvement in the convergence.

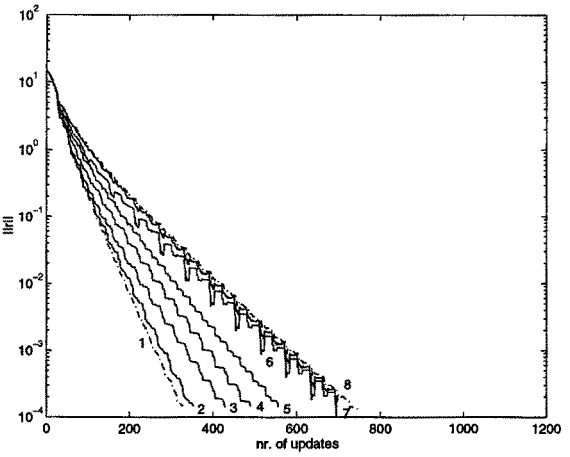

Fig. 1. Residual "fixed" for p updates. (1): Gauss-Seidel ($p = 1$) (2): $p = 2$ (3): $p = 3$ (4): $p = 6$ (5): $p = 10$ (6): $p = 15$ (7): $p = 30$ (8): Jacobi

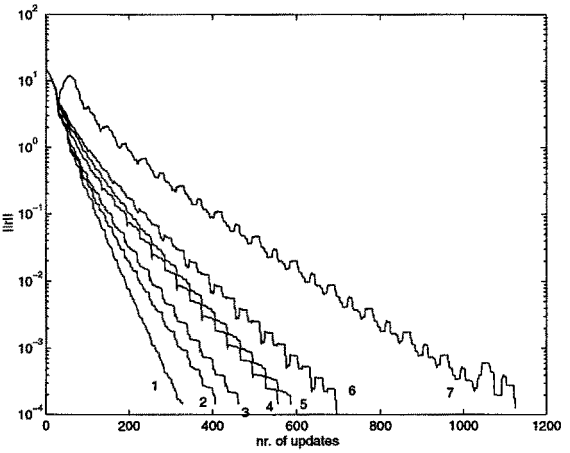

Fig. 2. "Time-lag" of p. (1): Gauss-Seidel ($p = 1$) (2): $p = 2$ (3): $p = 3$ (4): $p = 6$ (5): $p = 10$ (6): $p = 15$ (7): $p = 30$

The numerical testing depicted in Fig. 2 shows that increasing the number of processors means that older information is used to calculate any new iterate. This reduces the rate of convergence. To avoid that too old information is used to update the residual in the asynchronous implementation, we introduce a limit (Np) on the magnitude of time-lag, i.e., $t - Np \leq \tau_j^i(t) \leq t - 1$ for all i and j, and all $t \geq 0, t \in \mathcal{T}^i$. In [2], this is called a *partially asynchronous* iterative method. We use $p = g$ processors in a heterogeneous environment as in the former

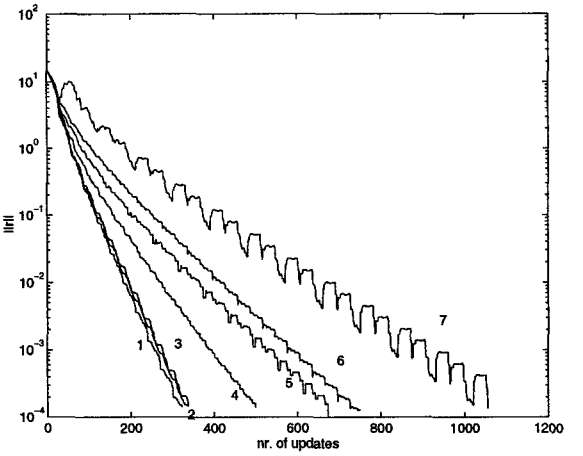

Fig. 3. Totally asynchronous on p processors. (1): $p = 6$ (2): $p = 3$ (3): Gauss-Seidel (4): $p = 10$ (5): $p = 15$ (6): Jacobi (7): $p = 30$

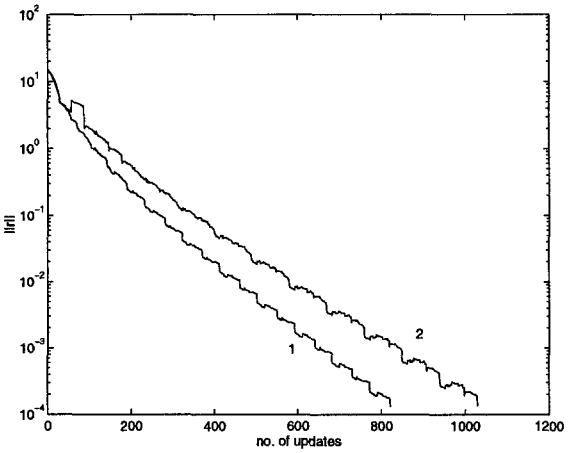

Fig. 4. Heterogeneous environment with $p = g$.
(1): "Wait-for-essential-neighbor" (WFE) (2): Totally asynchronous

experiment. When each processor has updated the residual a fixed number of times (N), we flush the queue at the master and broadcast the new residual. Letting $N = 1, 2, 3$, we observe that $N = 1$ gives (approximately) Jacobi's method and $N \geq 3$ gives (approximately) totally asynchronous method (Fig. 5).

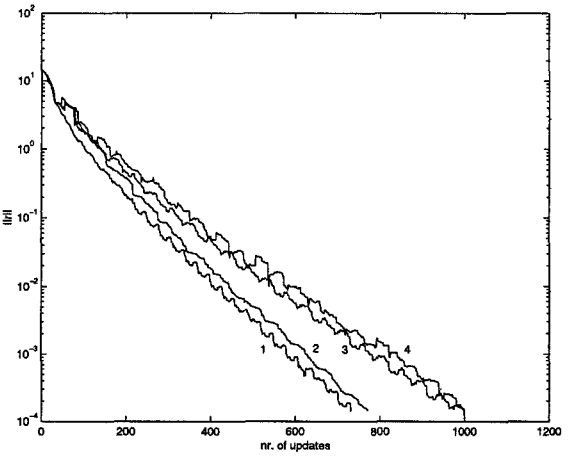

Fig. 5. Synchronization after N updates in a heterogeneous environment with $p = g$. (1): $N = 1$ ($\approx$ Jacobi) (2): $N = 2$ (3): Totally asynchronous (4): $N = 3$

5 Concluding Remarks

Elsner, Neumann and Vemmer [6] have proved, under certain assumptions, that increasing the number of processors decreases the convergence rate. This is shown in the implementation of asynchronous iterations on linear least squares problems and is explained as a result of using old information.

The heterogeneous environment had no significant effect on the rate of convergence. However, the changes in the residual is damped (compare curve 2 in Fig. 4 and curve 7 in Fig. 3).

The role of essential neighbors is also a factor that has to be taken into consideration. Blocks are forced to wait before receiving a new residual until a correction from their essential neighbors is received, and in the long run the time-lag is decreased. The result is an improvement in convergence and degradation in the deteriorations.

It is shown [4] that $\|r(t)\|_M$ is monotonically decreasing for the sequential case. In [12], the relaxation parameter ω_i is chosen such that $\|r(t)\|_M$ is minimized for every update. This reduces the effect of old information.

Another attempt to decrease deteriorations in the residual is the introduction of synchronization into the system. It is seen that synchronization after an *a priori* chosen number of corrections on the solution vector lessens the effect of old information and improves the convergence rate. However, the effect of the synchronization decreases rapidly with the "age" of the updates.

To our knowledge, the results of using asynchronous iterations on linear least squares problems have not been discussed in literature. There is no theory to characterize old information, only heuristics. A relaxation parameter can be used

to reduce the deteriorations. Synchronization is needed in many cases, but there is no theory to support *when* to synchronize. Synchronization after only one or two updates from each block reduces the effect of old information, but later synchronizations do not cure the deteriorations.

In [10, 12] and in this paper several issues were studied to reduce the deterioration in convergence. However, the single most influential factor based on the numerical testing is the existence of old information in the computation of the updates.

References

1. Baudet, G. M.: Asynchronous Iterative Methods for Multiprocessors. Journal of the ACM. **25** (1978) 226–244
2. Bertsekas, D. P., Tsitsiklis, J. N.: Parallel and Distributed Computation, Numerical Methods. Prentice-Hall Inc., Englewood Cliffs, N. J. (1989)
3. Björck, Å.: Numerical Methods for Least Squares Problems. SIAM, Philadelphia, PA (1996)
4. Dennis, J. E. Jr., Steihaug, T.: On the Successive Projections Approach to Least Squares Problems. SIAM J. Numer. Anal. **23** (1986) 717–733
5. Duff, I., Grimes, R. G., Lewis, J. G.: User's Guide for the Harwell-Boeing Sparse Matrix Collection. Technical Report TR/PA/92/86, CERFACS (1992)
6. Elsner, L., Neumann, M., Vemmer, B.: The Effect of the Number of Processors on the Convergence of the Parallel Block Jacobi Method. Lin. Alg. Appl. **154–156** (1991) 311–330
7. George, J. A., Heath, M. T.: Solution of Sparse Linear Least Squares Problems Using Givens Rotations. Lin. Alg. Appl. **34** (1980) 69–83
8. Kolm, P., Arbenz, P., Gander, W.: Generalized Subspace Correction Methods for Parallel Solution of Linear Systems. Technical Report TRITA-NA-9509, C2M2, Nada, KTH, Sweden (1995)
9. Savari, S. A., Bertsekas, D. P.: Finite Termination of Asynchronous Iterative Algorithms. *Parallel Computing.* **22** (1996) 39–56
10. Steihaug, T., Yalçınkaya, Y.: Asynchronous Methods and Least Squares: An Example of Deteriorating Convergence. Technical Report No. 131. Department of Informatics, University of Bergen, Bergen, Norway (1997)
11. Yalçınkaya, Y.: Asynchronous Solution of Linear Least Squares Problems Using Generalized Group Iterative Methods. Master's thesis. University of Bergen, Norway (1995)
12. Yalçınkaya, Y., Steihaug, T.: Asynchronous Methods and Least Squares: An Example of Deteriorating Convergence. Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics, August 24-29, 1997, Berlin, Germany (to appear) [part of [10]]