Asynchronous Methods and Least Squares: An Example of Deteriorating Convergence*

Trond Steihaug† and Yasemin Yalçınkaya‡
University of Bergen,
Department of Informatics,
Bergen, Norway

Abstract

We use block iterative methods for solving linear least squares problems. The subproblems are solved asynchronously on a distributed memory multiprocessor. It is observed that increased number of processors results in deteriorating convergence. We illustrate the deteriorating convergence by some numerical experiments. The deterioration of the convergence can be explained by contamination of the residual. The residual is contaminated by *old information*. Our purpose is to reduce the effect of old information. The issues investigated here are the effect of the number of processors, the role of *essential neighbors* [8] and heterogeneous processors. We include two heuristics to identify the information to be discarded and reduce the effect of old information: a relaxation factor and synchronization. The characterization of old information remains as an open problem.

1 Introduction

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

Due to the rapid development and increasing usage of parallel computers and distributed computing it has become important to adapt these 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 at all. This statement has led us to experiment with *totally asynchronous* [2] block iterative methods for the solution of linear least squares problems. We partition the columns of the coefficient matrix into (non)disjoint blocks of columns and then project the updated residual into each column subspace. This algorithm which is called *column oriented successive subspace correction* (CSSC) in [7], 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 and semi-normal equations 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. The processors

*This work has been partially supported by VISTA and Norwegian Research Council
†E-mail:Trond.Steihaug@ii.uib.no
‡E-mail:yasemin@ii.uib.no

use the residual available at the start of their computations without waiting for 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 [10]. 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. Our purpose is to reduce the effect of old information, thus decreasing the deteriorations in convergence.

In the following, we first give the framework of block iterative method for the linear least squares problem and state the sequential algorithm. We also mention an example problem that makes use of successive solution of linear least squares problem. The subproblems in this algorithm are to be solved using a direct method. We give a short discussion of factorization techniques in this context. Then, we introduce the totally asynchronous algorithmic model and the algorithm for the asynchronous implementation of our method. In Section 5, the results of our experiments are presented. It is followed by a section on open problems and concluding remarks.

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

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

$$\min\{\|A_1x_1 + A_2x_2 + \cdots + A_gx_g - b\|_M : x_1 \in \mathbb{R}^{n_1}, x_2 \in \mathbb{R}^{n_2}, \ldots, x_g \in \mathbb{R}^{n_g}\}\ . \tag{2}$$

We will later allow for A_i to share columns with A_j, i.e., overlapping column blocks.

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

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

Consider:

$$\begin{aligned} \textstyle\sum_{j=1}^{i-1} A_j x_j^{k+1} + A_i x_i^{k+1} \quad &+ \quad \textstyle\sum_{j=i+1}^{g} A_j x_j^k - b \\ &= \quad A_i(x_i^{k+1} - x_i^k) + \textstyle\sum_{j=1}^{i-1} A_j x_j^{k+1} + \sum_{j=i}^{g} A_j x_j^k - b \\ &= \quad A_i s_i^k + r^{k+(i-1)/g}, \end{aligned}$$

where $s_i^k = x_i^{k+1} - x_i^k$ is the step or correction, and

$$r^{k+(i-1)/g} = \sum_{j=1}^{i-1} A_j x_j^{k+1} + \sum_{j=i}^{g} A_j x_j^k - b \tag{3}$$

is the residual. The least squares subproblem, now, is:

Solve for $s_i^k \in \mathbb{R}^{n_i} : \min\{\|A_i s_i + r^{k+(i-1)/g}\|_M\}$.

The residual can be shown to satisfy

$$r^{k+i/g} = r^{k+(i-1)/g} + A_i s_i^k \ .$$

The new approximate solution is

$$x_i^{k+1} = x_i^k + s_i^k, \quad i = 1, \ldots, g \ .$$

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

$$x^{k+1} = \sum_{i=1}^{g} \bar{x}_i^{k+1} \ .$$

Introducing $\hat{r}$ we get:

for $i = 1, 2, \ldots, g$ **do**
$\hat{r} = r^{k+(i-1)/g}$.
Solve for s_i^k : $\min\{\|A_i s_i + \hat{r}\|_M : s_i \in \mathbb{R}^{n_i}\}$. (4)
Update the residual: $r^{k+i/g} = r^{k+(i-1)/g} + A_i s_i^k$.
Update the solution: $x^{k+i/g} = x^{k+(i-1)/g} + \bar{s}_i^k$.

We have a block Gauss-Seidel iteration on the normal equations for (1). When we use the residual as $\hat{r} = r^k$, we get Jacobi iteration. 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.

The formulation above has a direct extension to overlapping column blocks. Let $\mathbf{n} = n_1 + \ldots n_g$, where $A_i \in \mathbb{R}^{n_i}$ and $\mathbf{A} = (A_1|A_2|\ldots|A_g), \mathbf{A} \in \mathbb{R}^{m \times \mathbf{n}}$. The new m by $\mathbf{n}$ least squares problem can be written as

$$\min_{\mathbf{x} \in \mathbb{R}^{\mathbf{n}}} \|\mathbf{A}\mathbf{x} - b\|_M,$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_g)^T$, $\mathbf{x} \in \mathbb{R}^{\mathbf{n}}$.

We can now define the block SOR algorithm for the solution of (1).

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

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

Applications of least squares arise in a great number of fields. Efficient methods for the solution of weighted least squares problems are becoming increasingly important. One of those fields that need the successive solution of linear least squares problems is the primal dual interior point method. We define this problem next.

2.1 Successive Solution of Linear Least Squares Problems

Let $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}$. The linear programming problem

$$\begin{array}{rc} \text{minimize} & c^T x \\ \text{s.t.} & Ax = b \\ & x \geq 0, \end{array}$$

has its dual in the form

$$\begin{array}{rc} \text{maximize} & b^T y \\ \text{s.t.} & A^T y + z = c \\ & z \geq 0, \end{array}$$

where $y \in \mathbb{R}^m$ is the dual variable and $z \in \mathbb{R}^n$ is the dual slack. A primal dual interior point method for this problem will attempt to solve the following Newton step equation [9]

$$\begin{pmatrix} 0 & A^T & I \\ A & 0 & 0 \\ Z & 0 & X \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta z \end{pmatrix} = \begin{pmatrix} -r_c \\ -r_b \\ -XZe + \sigma\mu e \end{pmatrix} \tag{5}$$

where X =diag(x), Z =diag(z), $r_b = Ax - b$, $r_c = A^T y + z - c$, μ is the duality gap, σ is an algorithm-dependent parameter between [0,1], and e is the vector of all ones.

Equation (5) can be reformulated eliminating Δz to give

$$\begin{pmatrix} X^{-1}Z & A^T \\ A & \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = \begin{pmatrix} c - A^T y - \sigma\mu X^{-1} e \\ Ax - b \end{pmatrix}, \tag{6}$$

$$\Delta z = X^{-1}(\sigma\mu e + Z\Delta x) - z,$$

which is known as the *augmented system.*

Let $M = (X^{-1}Z)^{-1}$. Then, from (6), we get

$$AMA^T \Delta y = AM(c - A^T y - \sigma\mu X^{-1} e) + b - Ax,$$

and

$$\Delta x = M(c - A^T(y + \Delta y) - \sigma\mu X^{-1} e) \ .$$

We need to solve a (weighted) linear least squares problem:

$$\min \|A^T \Delta y - d\|_M,$$

where

$$d = c - A^T y - \sigma\mu X^{-1} e + M^{-1}(x - \hat{x}),$$

and $\hat{x}$ is any feasible point $A\hat{x} = b$.

3 Factorization

In (4), the system to be solved is substantially smaller than the original problem. This enables us to use a direct method in our computation for s_i^k, and brings out the need for a good factorization of A_i. The criteria for the best factorization algorithm in our case is that the algorithm should store and operate only on the nonzero entries of the matrix, should try to minimize the creation of new nonzeros as computations proceed, and the trade-offs between sparsity of the system and numerical stability should be taken into consideration. Here, we will consider three methods: Method of normal equations, QR factorization and semi-normal equations (SNE).

In the method of normal equations we form the matrix $C = A^T A$ and apply Cholesky algorithm to compute R such that $C = R^T R$. This transfers the system of normal equations $A^T Ax = A^T b$ to $R^T Rx = A^T b$, with R upper triangular, so that the system can be solved by forward and backward substitution. When the same system is to be solved for several right-hand sides, we need to store matrices R and A.

For well-conditioned problems the method of normal equations is quite satisfactory. However, for ill-conditioned or stiff problems this method may lead to substantially less accurate solutions than methods based on the QR decomposition.

The potential numerical instability of the method of normal equations is due to loss of information in explicitly forming $A^T A$ and $A^T b$, and to the fact that the condition number of $A^T A$ is the square of that of A. Orthogonalization methods avoid both of these sources of inaccuracy by working directly with A. An orthogonal matrix $Q \in \mathbb{R}^{m\times m}$ is computed which reduces $[A, b]$ to the form

$$Q^T A = \begin{bmatrix} R \\ 0 \end{bmatrix}, \; Q^T b = \begin{bmatrix} c \\ d \end{bmatrix},$$

where $R \in \mathbb{R}^{n\times n}$ and $c \in \mathbb{R}^n$. Matrix R can be computed row sequentially using Givens rotations. Givens rotations should be applied simultaneously to b to form $Q^T b$. In the implementation by George and Heath [6] the Givens rotations are not stored but discarded after use. Hence, only enough storage to hold the final R and a few extra vectors for the current row and right-hand side is needed in the main memory during the factorization phase.

Discarding Q creates a problem since we wish to solve additional problems having the same matrix A but different right-hand sides b that are not known at the time of the QR factorization. If the original matrix A is saved in addition to R, one can use the semi-normal equations (SNE)

$$R^T Rx = A^T b \ .$$

This approach is intermediate in numerical stability between normal equations and orthogonalization [3].

4 Parallelization

In this section, we will 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 the (dis)advantages 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 now the solution of (4). 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 current iteration before starting with the next iteration. 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. But we can do better. In synchronized algorithms, the faster processors waiting for the slower ones to complete their computations before they can enter the critical section 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 can potentially reduce the synchronization penalty caused by fast processors waiting for slow processors to complete their computations, and for slow communication channels to deliver messages. The reason is that processors can execute more iterations when they are not constrained to wait for the results of the computation on other processors. In the computation of an iterate, nothing is imposed on the use of the values of the previous iterates. The only thing that is required is that, eventually, the values of an early iterate cannot be used any more in further evaluations. This condition is met as long as no processor falls out of the system. However, removing restrictions on the former iterates used in computations brings out the danger that the iterations are performed on the basis of outdated (old) information, and their results will not be effective. We will consider some heuristic experiments in Section 5 and 6 to decrease the effect of old information in the system.

An important disadvantage of asynchronism is that it can destroy 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. In all cases, the analysis of asynchronous algorithms is considerably more difficult than for their synchronous counterparts. Necessary conditions for the convergence of linear problems is given by Bertsekas and Tsitsiklis in [2]. They also provide a rate of convergence analysis and a comparison between synchronous and asynchronous algorithms.

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

4.1 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_i in (3). 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:

$$x_i(t) = x_i(t-1), \ t \notin \mathcal{T}^i,$$

or

$$x(t) = x(t-1) + \bar{s}_i(t) \ .$$

4.2 Asynchronous Implementation

We will now give the main algorithm for 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, and **Gsum** is a global vector sum operation executed on all the processors.

Algorithm 2

Initialization:

  Subdivide A into g blocks.

  Choose $0 < \omega_i < 2, i = 1, 2, \ldots, g$.

  Choose $x_i(0), i = 1, \ldots, g,\ x(0) = \sum_{i=1}^{g} \bar{x}_i(0)$.

  Compute $r(0) = Ax(0) - b$.

  Initiate each processor $i = 1, \ldots, g$:

    Receive(A_i, p_0).

    Receive$(\bar{x}_i(0), p_0)$.

    Receive(ω_i, p_0).

    Preprocess.

  Broadcast$(r(0))$.

Loop :

Let t be a global counter of corrections, and let t_1^i and t_2^i be two consecutive elements in $\mathcal{T}^i$.

  $t = 0$.

  while not *termination* **do**

    if *slave* **then**

      Solve for s_i :

        $\min\{\|A_i s_i + r(t_1^i)\|_M : s_i \in \mathbb{R}^{n_i}\}$.

      Update $\bar{x}_i : \bar{x}_i = \bar{x}_i + \omega_i \bar{s}_i$.

      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 $t_1^i\}$

      $r(t+1) = r(t) + \omega_i A_i s_i$.

      Check for *termination*.

      if *termination* **then**

        Gsum$(\bar{x}_i)$.

      else

        $t = t + 1$;

        Send$(r(t), p_i)$. $\{t_2^i = t\}$

5 Experiments

In this section we will present the results of some numerical experiments. The first four experiments illustrate the deteriorating convergence when one passes from sequential to parallel implementation, and from few processors to many. The next two experiments aim at decreasing the contamination of the residual.

Test problems used in the experiments are taken from Harwell-Boeing sparse matrix test collection. Groups are formed by taking blocks of consecutive columns. If not otherwise stated 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.

Before the asynchronous implementations, in the first two experiments, we simulate the behavior of the residual on the master processor for different number of slaves where $g \geq p$. A random block order is used for the updates in the simulation giving an effect of asynchronism.

We first keep the residual "fixed" for p-updates, i.e., $\hat{r} = r^{k+c/g}$ in (4) and $c = p(i//p)$. Here, c is a counter and we assume for simplicity that $g \bmod p = 0$. This routine simulates Gauss-Seidel sweeps with Jacobi iteration on blocks of g/p block components. When $p = g$, we get pure Jacobi. We stated in Section 2 that in Algorithm 1 we have 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 result. We see in Figure 1 that the resulting residual from the experiment 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, we keep a "time lag" of p, where $\hat{r} = r^{k+(i-p)/g}$ in (4). This is the simulation of asynchronous Gauss-Seidel method on p processors, where the processors update the residual on the master in a fixed order. The same block assignment as in the former case is used. Again, increased number of processors result in increased deterioration as seen in Figure 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. A minor modification of Elsner, Neumann and Vemmer [5] gives

$$\rho(p+1) \geq \rho(p) \ .$$

Figure 3 depicts the main result of [10]. Here, a totally asynchronous implementation on $p = g$ processors is compared to sequential Gauss-Seidel and Jacobi methods. The figure shows the deterioration of convergence when one moves from sequential Gauss-Seidel and Jacobi to totally asynchronous implementation.

To confirm the simulation result from the second experiment we implement totally asynchronous iterations on different number of processors p, $p \leq g$ and $g \bmod p = 0$. We see in Figure 4 that for some 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. This is in accordance with the simulation results in Figure 2.

The above experiments are done on homogeneous processors where the computing speed of each processor is nearly the same. When we use heterogeneous processors with computing speeds varying between approximately 1-7 the deterioration in the residual is damped (Figure 5). However, the improvement in the convergence rate is not significant.

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* [8] of block i. Let $t_1, t_2 \in \mathcal{T}^i$ be consecutive times of update from block i. When we compute

$$\min\{\|A_i s_i + r(t_1))\|_M\},$$
$$r(t_2) = r(t_2 - 1) + \omega_i A_i s_i$$

at time t_2, $A_i^T r(t_2) = 0$ 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. In Figure 6, the curve marked WFE illustrates the implementation where the processors wait for their essential neighbors. We see a decrease in the deteriorations and as a result an improvement in the convergence.

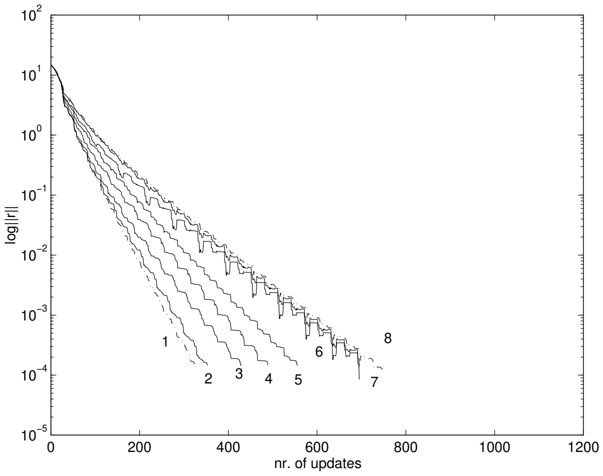

Figure 1: Residual "fixed" for p updates. (1): Gauss-Seidel ($p = 1$) (2): $p = 2$ (3): $p = 3$ (4): $p = 6$ (5): $p = 10$ (6): $p = 15$ (7): $p = 30$ (8): Jacobi

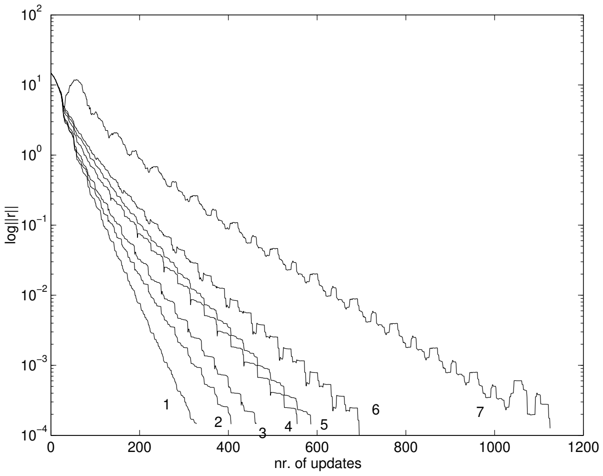

Figure 2: "Time lag" of p. (1): Gauss-Seidel ($p = 1$) (2): $p = 2$ (3): $p = 3$ (4): $p = 6$ (5): $p = 10$ (6): $p = 15$ (7): $p = 30$

6 Open Problems

6.1 The Effect of Synchronization

Elsner, Neumann and Vemmer [5] showed that increasing the number of processors means that older information is used to calculate any new iterate. This reduces the rate of convergence. They assume that the order of the blocks is fixed. In their experiment there is a time-lag of p on the updates. 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. When each processor has updated the residual a fixed number of times (N), we flush the queue at the master and broadcast the

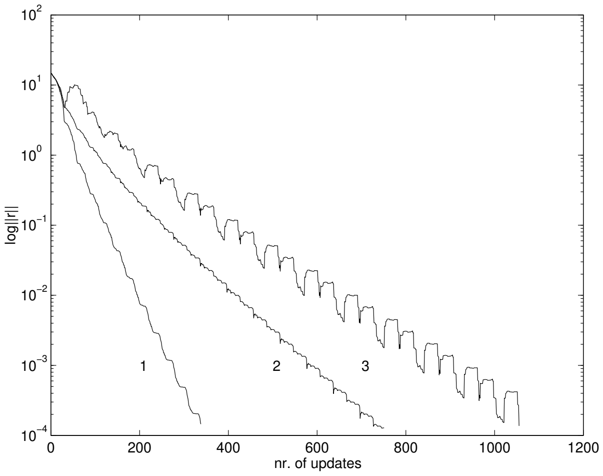

Figure 3: Sequential versus totally asynchronous. (1): Gauss-Seidel (2): Jacobi (3): Totally asynchronous

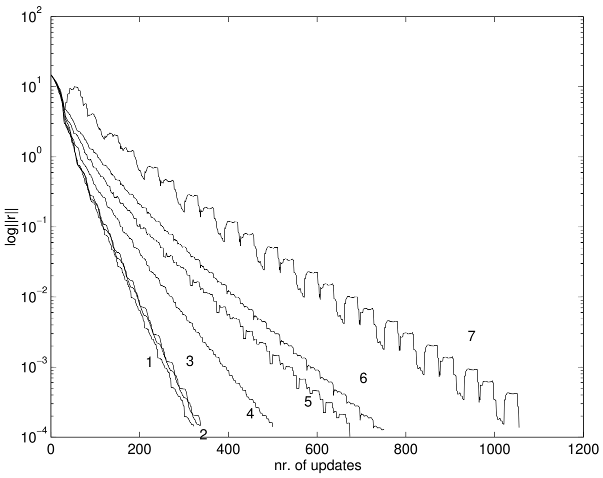

Figure 4: Totally asynchronous on p processors. (1): $p = 6$ (2): $p = 3$ (3): Gauss-Seidel (4): $p = 10$ (5): $p = 15$ (6): Jacobi (7): $p = 30$

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 (Figure 7).

6.2 The Effect of a Relaxation Parameter

Let t_1 and t_2 be two consecutive updates of block 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:

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

For $t_2 \gg t_1$ old information is contaminating the residual.

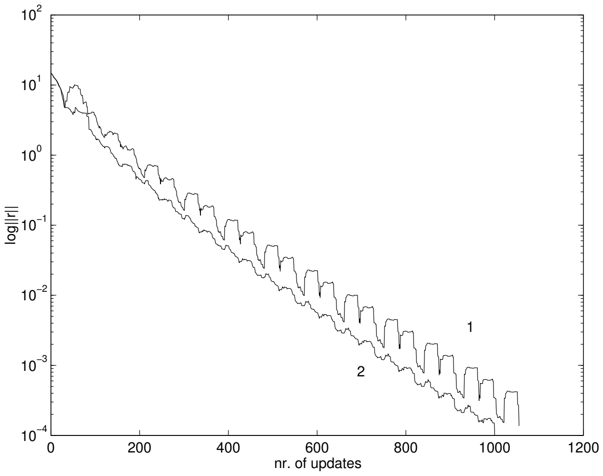

Figure 5: Homogeneous versus heterogeneous processors. (1): $p = 30$ identical processors (2): $p = 30$ processors with computing speed varying with 1-7

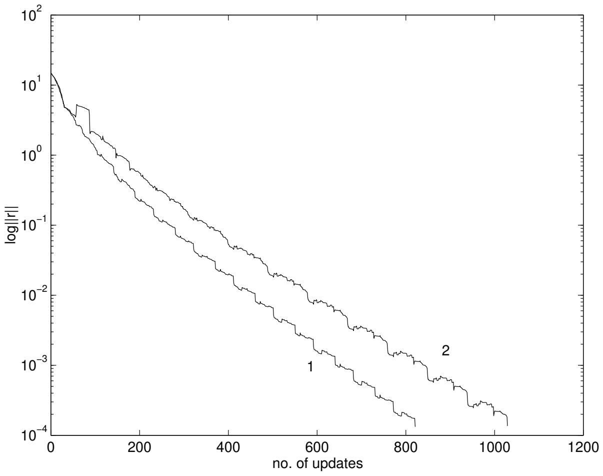

Figure 6: "Wait-for-essential-neighbor"(WFE) versus totally asynchronous. (1): WFE on $p = 30$ processors (2): Totally asynchronous on $p = 30$ processors

It is observed in the sequential case that the residual $\|r(t)\|_M$ is monotonically decreasing for SOR method, and $\|r(t)\|_M \le \|r(t-g)\|_M$ for Jacobi [4]. In our experiment we choose ω_i such that $\|r(t_2)\|_M$ is minimized. The values of s_i that give $(A_i s_i)^T M r \ge 0$ are discarded. Otherwise, ω_i is chosen as

$$\omega_i = \frac{-(A_i s_i)^T M r}{\|A_i s_i\|_M^2}, \quad 0 < \omega_i < 2 \ .$$

As seen in Figure 8, this choice for ω_i reduces the effect of old information on the residual.

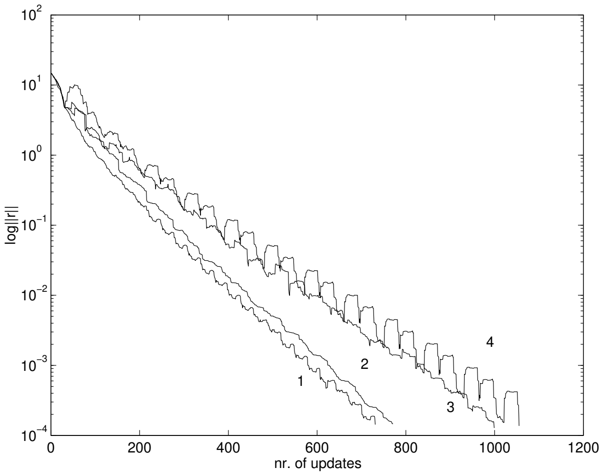

Figure 7: Synchronization after N updates versus totally asynchronous.
(1): $N = 1$ ($\approx$ Jacobi) (2): $N = 2$ (3): $N = 3$ (4): Totally asynchronous

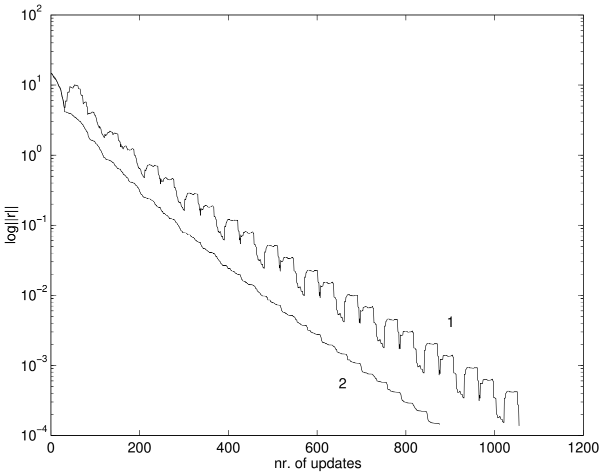

Figure 8: Asynchronous SOR. (1): $\omega = 1.0$, (2): Line search

7 Concluding Remarks

Elsner, Neumann and Vemmer [5] have proved, under certain assumptions, that increasing the number of processors decreases the convergence rate, since the effect of old information is increased. This is shown in our implementation of asynchronous iterations on linear least squares problems.

We do some experiments to illustrate the deteriorating convergence, and attempt to decrease the effect of old information. First, we check the effect of heterogeneous processors. We find out that though the deterioration of convergence is damped, the damping effect is not significant.

The role of essential neighbors is also a factor that has to be taken into consideration. In the least squares problems the set $E_i = \{j \mid$ block i and block

j have nonzero elements on the same row positions$\}$ is a characterization of *essential neighbors* of block i. Let t_1, t_2 be consecutive times of update from block i. When we compute a correction on block i using the global residual at time t_1 and update the residual again at time t_2, the correction computed using this new residual will be zero unless a block $j, j \in E_i$ has sent a correction between t_1 and t_2. Groups are thus forced to wait before receiving a new residual until a correction from their essential neighbors is received. The result is an improvement in convergence and degradation in the deteriorations.

We want to determine the data that should be discarded. In order to achieve this, we choose ω as a line search and prevent the contamination of the residual by old data.

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. We have introduced two open problems: the effect of a relaxation parameter and synchronization. So far there is no theory to characterize old information, only heuristics. A relaxation parametercan 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 has a damping effect on the old information, but later synchronizations do not cure the deteriorations.

References

[1] G. M. Baudet, *Asynchronous Iterative Methods for Multiprocessors*, J. of the ACM 25, pp. 226–244 (1978).

[2] D. P. Bertsekas, J. N. Tsitsiklis, Parallel and Distributed Computation, Numerical Methods, Prentice-Hall Inc., Englewood Cliffs, NJ (1989).

[3] Å. Björck, Numerical Methods for Least Squares Problems, SIAM, Philadelphia, PA (1996).

[4] J. E. Dennis, Jr., T. Steihaug, *On the Successive Projections Approach to Least Squares Problems*, SIAM J. Numer. Anal 23, pp. 717–733 (1986).

[5] L. Elsner, M. Neumann, B. Vemmer, *The Effect of the Number of Processors on the Convergence of the Parallel Block Jacobi Method*, Lin. Alg. Appl 154–156, pp. 311–330 (1991).

[6] J. A. George, M. T. Heath, *Solution of Sparse Linear Least Squares Problems Using Givens Rotations*, Lin. Alg. Appl. 34, pp. 69–83 (1980).

[7] P. Kolm, P. Arbenz, W. Gander, *Generalized Subspace Correction Methods for Parallel Solution of Linear Systems*, Tech. Rep. TRITA-NA-9509, C2M2, Nada, KTH, Sweden (1995).

[8] S. A. Savari, D. P. Bertsekas, *Finite Termination of Asynchronous Iterative Algorithms*, Parallel Computing 22, pp. 39–56 (1996).

[9] S. J. Wright, Primal-Dual Interior-Point Methods, SIAM, Philadelphia, PA (1997).

[10] Y. Yalçınkaya, *Asynchronous Solution of Linear Least Squares Problems Using Generalized Group Iterative Methods*, Master's thesis, University of Bergen, Norway (1995).