Improved parallel QR method for large least squares problems involving Kronecker products

Donald W. Fausett, Charles T. Fulton*, Hany Hashish

Department of Applied Mathematics, Florida Institute of Technology, Melbourne, FL 32901-6988, USA

Received 31 January 1996; revised 20 June 1996

Abstract

A new algorithm is presented for the efficient solution of large least squares problems in which the coefficient matrix of the linear system is a Kronecker product of two smaller dimension matrices. The solution algorithm is based on QR factorizations of the smaller dimension matrices. Near perfect load balancing is achieved by exploiting a 'commutativity' property of the Kronecker product, and communication requirements are minimized by employing a binary exchange algorithm for matrix transposition. The parallel algorithm is presented, and timing results are shown from test runs on an Intel i860 computer.

Keywords: Kronecker product; Overdetermined least squares; QR factorization; Matrix algorithms; Parallel processing

1. Introduction

In this paper we present a parallel algorithm for solution of the (full rank) least squares problem

\[(A \otimes B)x = t,\]

where \(A \in M_{m,p}\) and \(B \in M_{n,q}\), with \(\text{rank}(A) = p\), \(\text{rank}(B) = q\), and \(x \in \mathbb{R}^p\), \(t \in \mathbb{R}^m\).

In our previous paper [12], we presented a new method for solving (1.1) based on QR decompositions of \(A\) and \(B\). A parallel algorithm was also outlined. Here we describe a modified algorithm which avoids the need to compute the inverse of one of the \(R\)-matrices, and discuss its implementation on the Intel i860 computer. The modified algorithm makes use of a 'commutativity' property of the Kronecker product. A pair of upper triangular systems is produced from which the least squares solution can be obtained by backsolving the two systems in parallel. This results in better

* Corresponding author. E-mail: fulton@zach.fit.edu.

1 This work was supported by National Science Foundation Grant ASC-8821626, New Technologies Program, Division of Advanced Scientific Computing.

0377-0427/97/$17.00 © 1997 Elsevier Science B.V. All rights reserved

PII S0377-0427(96)00109-4
computational load balancing and significantly less communication overhead than the version of the
given in [12].
In this paper we shall need to make use of standard properties of the Kronecker product [14, 21, 23, 27]. In addition to those properties cited in [12] we shall require:

**Property (i).** For all $A \in M_{m,p}$, $B \in M_{n,q}$ and $X \in M_{q,p}$ we have
\[(A \otimes B)\text{vec}(X) = \text{vec}(B \cdot X \cdot A^T),\] (1.2)
where $\text{vec}(X)$ [19] denotes the vector obtained by stacking the columns of $X$.

**Property (ii).** If $A \in M_{m,p}$ and $B \in M_{n,q}$ then the least squares problem
\[(A \otimes B)x = t\] (1.3a)
has the equivalent matrix formulation
\[B \cdot X \cdot A^T = T,\] (1.3b)
where $x = \text{vec}(X)$ and $t = \text{vec}(T)$.

**Property (iii) (Commutativity).** For all $A \in M_{m,p}$, $B \in M_{n,q}$, $x \in \mathbb{R}^{pq}$ and $t \in \mathbb{R}^{mn}$, we have
\[(A \otimes B)x = t \quad \text{if and only if} \quad (B \otimes A)x_T = t_T,\] (1.4a)
where $x = \text{vec}(X)$, $x_T = \text{vec}(X^T)$, $t = \text{vec}(T)$ and $t_T = \text{vec}(T^T)$. Using (1.2) the equivalent matrix form of (1.4a) is readily seen to be
\[B \cdot X \cdot A^T = T \quad \text{if and only if} \quad A \cdot X^T \cdot B^T = T^T,\] (1.4b)
a result which follows by taking the transpose of both sides.
For Properties (i) and (ii) see [19, 20, 23, 27].

*Note:* Since $A \otimes B \neq B \otimes A$, the above “commutativity” property has no algebraic significance, but can be regarded as a convenient way of interchanging the order of $A$ and $B$ in the Kronecker product, with an explicit description of the resulting effect on the ordering of the components of $x$. Algebraically, one does, however, have the existence of permutation matrices $P_1 \in M_{mn,nn}$ and $P_2 \in M_{pq,pq}$ such that $A \otimes B = P_1(B \otimes A)P_2$ [20, 23]. It is also noteworthy that when $A$ and $B$ are square, $P_1$ and $P_2$ are so-called “stride permutation” matrices [15].

A modification of the least squares QR-method from [12] is described in Section 2. The implementation of the parallel algorithm is described in Section 3. The new algorithm makes use of the $Q$ and $R$ matrices from Householder QR-factorizations (using Householder reflections) of the $A$ and $B$ matrices; it therefore inherits the numerical stability associated with the Householder QR factorizations. Section 4 discusses the computational complexities of the modified algorithm, and Section 5 presents timing and speed-up data from test runs on an Intel i860 computer. For information on applications of the Kronecker product least squares problem in many diverse areas of science and engineering we refer to [1, 3, 10, 22, 28, 29, 31, 32].

The present algorithm has also recently been extended by Hashish [17] to handle the case when the ranks of $A$ and $B$ are deficient, i.e. $\text{rank}(A) < p$ and $\text{rank}(B) < q$. 
2. A Kronecker product least squares algorithm

Suppose that the full rank matrices $A$ and $B$ have each been QR decomposed with column pivoting [13, 16], so that

$$BP_1 = Q_1 R_1 = Q_1 \begin{pmatrix} R^{(1)} \\ O^{(1)} \end{pmatrix} \quad \text{and} \quad AP_2 = Q_2 R_2 = Q_2 \begin{pmatrix} R^{(2)} \\ O^{(2)} \end{pmatrix},$$

where

$$Q_1 \in \mathbb{M}_{m,n}, \quad Q_2 \in \mathbb{M}_{m,m}$$

are orthogonal matrices,

$$R_1 \in \mathbb{M}_{n,q} \quad \text{and} \quad R_2 \in \mathbb{M}_{m,p}$$

are upper triangular matrices,

$$R^{(1)} \in \mathbb{M}_{q,q} \quad \text{and} \quad R^{(2)} \in \mathbb{M}_{p,p}$$

are square, upper triangular matrices,

$$O^{(1)} \in \mathbb{M}_{n-q,q} \quad \text{and} \quad O^{(2)} \in \mathbb{M}_{m-p,p}$$

are zero matrices,

and $P_1 \in \mathbb{M}_{n,q}$ and $P_2 \in \mathbb{M}_{m,p}$ are permutation matrices arising from the column pivoting used to keep the diagonal elements as far away from zero as possible.

In [12] it was shown that substitution of (2.1) into (1.1) converts the least squares problem to the equivalent form

$$P_3(R_2 \otimes R_1)y = \begin{pmatrix} R^{(2)} \otimes R^{(1)} \\ O^{(3)} \end{pmatrix} y = P_3(Q_2^T \otimes Q_1^T)t,$$

where $y = (P_2^T \otimes P_1^T)x$. Here the permutation matrix $P_3$ is defined so as to make the first equality in (2.2) hold with zero matrix $O^{(3)} \in \mathbb{M}_{m-n, pq}$.

If we introduce the partitioning of the orthogonal matrices $Q_1$ and $Q_2$,

$$Q_1 = (Q_1^{(1)}, Q_1^{(2)}) \quad \text{and} \quad Q_2 = (Q_2^{(1)}, Q_2^{(2)}),$$

where $Q_{1}^{(1)} \in \mathbb{M}_{n,q}$ and $Q_{1}^{(2)} \in \mathbb{M}_{m,m}$, then

$$P_3(Q_2^T \otimes Q_1^T) = \begin{pmatrix} Q_2^{(2)}^T \otimes Q_1^{(1)}^T \\ Q_1^{(2)}^T \otimes Q_2^{(1)}^T \\ Q_2^{(1)}^T \otimes Q_1^{(2)}^T \\ Q_2^{(2)}^T \otimes Q_1^{(2)}^T \end{pmatrix}.$$  

Let

$$h = (Q_1^{(2)^T} \otimes Q_1^{(1)^T})t = [P_3(Q_2^T \otimes Q_1^T)t]_{pq},$$

where $[P_3(Q_2^T \otimes Q_1^T)t]_{pq}$ denotes the first $pq$-components of the right-hand side of (2.2), cf. [12]. From the equivalence of the Kronecker product relation (1.3a) with its matrix counterpart (1.3b), every Kronecker product equation may be written in an equivalent matrix form. The equivalent matrix form of Eq. (2.5a) is

$$H = Q_1^{(1)^T} T Q_1^{(2)},$$

where $t = \text{vec}(T)$ and $h = \text{vec}(H)$, and $H \in \mathbb{M}_{q,p}$.

Putting (2.4) on the right-hand side of (2.2), it follows that the least squares solution of (2.2) (or, equivalently, the least squares solution of the original problem (1.3)) is the exact solution of the square nonsingular system

$$(R^{(2)} \otimes R^{(1)})y = h,$$
or
\[ R^{(1)} Y R^{(2)\dagger} = H, \quad (2.6b) \]
where \( y = \text{vec}(Y) \) and \( h = \text{vec}(H) \), and \( Y \in M_{q,p} \). For clarity we shall list both the Kronecker product and matrix versions of all equations which follow in this section.

Combining (2.5) and (2.6) the solution \( y \) of the least squares problem (2.2) may be written as
\[ y = (R^{(2)} \dagger \otimes R^{(1)} \dagger) (Q_{1}^{(2)} \otimes Q_{1}^{(1)} \dagger) h, \quad (2.7a) \]
or
\[ Y = R^{(1)} \dagger Q_{1}^{(1)} \dagger T Q_{1}^{(2)} R^{(2)} \dagger, \quad (2.7b) \]
which indicates the dependence of the solution on the QR factorizations of \( A \) and \( B \). As we shall see, the manner in which computations implementing (2.7) are parallelized plays a significant role in the computational efficiency on a parallel computer.

We now describe our modification of the algorithm given in [12]. Using the matrix multiplication property for Kronecker products, Eq. (2.6), may be written as
\[(R^{(2)} \otimes I_{1}) \cdot (I_{2} \otimes R^{(1)}) y = h, \quad \text{or} \quad I_{1} (R^{(1)} Y I_{2}) R^{(2)\dagger} = H, \]
where \( I_{1} \in M_{q,q} \) and \( I_{2} \in M_{p,p} \) are identity matrices. Putting
\[ z = (I_{2} \otimes R^{(1)}) y \quad \text{or} \quad Z = R^{(1)} Y, \]
where \( z = \text{vec}(Z) \), \( Z \in M_{q,p} \), we may write the nonsingular system (2.6) in the form of two uncoupled systems of equations
\[ (R^{(2)} \otimes I_{1}) z = h \iff (I_{1} \otimes R^{(2)}) z_{T} = h_{T} \quad (2.8a) \]
or
\[ I_{1} Z R^{(2)\dagger} = H \iff R^{(2)} Z^{T} = H^{T}, \quad (2.8b) \]
and
\[ (I_{2} \otimes R^{(1)}) y = z \quad \text{or} \quad R^{T} Y = Z. \quad (2.9a,b) \]

Accordingly, we may compute \( h_{T} \) from the \( Q \)-matrices using (1.4a) and (2.5a), solve the system (2.8) for \( z_{T} \) (or \( Z^{T} \)) by backsubstitution, redistribute \( z_{T} \) to get \( z \) (that is, perform a matrix transpose on \( Z^{T} \) to get \( Z \)), and then solve the system (2.9) by backsubstitution.

The algorithm given in [12, p. 224] differs from this in that Eq. (2.8a) is solved for \( z \) by computing
\[ z = (R^{(2)} \dagger \otimes I_{1}) h \quad \text{or} \quad Z = HR^{(2)\dagger}, \quad (2.10a,b) \]
and then the perfectly parallel equation
\[ (I_{2} \otimes R^{(1)}) y = z \quad (2.11) \]
is solved by backsubstitution.
The main advantage of the modified algorithm (2.8), (2.9) over the previous algorithm (2.10), (2.11) is that (2.8a) is written with $R^{(2)}$ appearing on the right-hand side of the Kronecker product (or with $R^{(2)}$ appearing on the left side of the matrix product in (2.8b)), so that both (2.8) and (2.9) have the perfectly parallel form of Eq. (4.3) in [12]. Therefore there is no need to compute $R^{(2)-1}$. This also represents an improvement of the algorithm in that floating point operations are reduced. The computation of the inverse of the upper triangular matrix $R^{(2)}$ requires $p^3 + \frac{1}{2} p$ flops, and the resulting matrix product $HR^{(2)-1}$ in (2.10) requires an additional $p^2 q$ flops for a total of $p^2 q + \frac{1}{2} p^3 + \frac{1}{3} p$ flops, compared to only $p^2 q$ flops for the solving of the $q$-systems of Eqs. in (2.8) by backsubstitution.

With respect to parallel implementation, the algorithm (2.8)(2.9) is also more advantageous in that better load balancing may be achieved, and with less communication overhead. Namely, the right-hand side vector $h_T$ (or matrix $H^T$) in Eq. (2.8) is generated from (2.5) and then the two stages of backsolves indicated in Eqs. (2.8) and (2.9) are performed in parallel by distributing the right-hand sub-vectors of $h_T$ and $z$, respectively (or by distributing the columns of $H^T$ and $Z$, respectively) as equally as possible across the processors. As a consequence, the load balancing for both sets of backsolves is very good since each processor is performing the same number of backsolves. Moreover, the algorithm (2.8), (2.9) has the advantage that all those subvectors of $h^T$ (those columns of $H^T$) which are assigned to a given processor for the backsolves in (2.8) may be computed on the same processor, thus eliminating all communication overhead in passing from (2.5) to (2.8). For we may write Eq. (2.5) using Property (iii) and the matrix multiplication property for Kronecker products as

\[
h_T = (Q_1^{(1)\top} \otimes Q_1^{(2)\top}) t_T = (I_1 \otimes Q_1^{(2)\top})(Q_1^{(1)\top} \otimes I_3) t_T
\]

(2.12a)

or

\[
H^T = Q_1^{(2)\top} [I_3 T^T Q_1^{(1)}] I_1,
\]

(2.12b)

where $h_T = vec(H^T)$, $t_T = vec(T^T)$, and $I_1 \in M_{m,m}$, and compute those subvectors of $h_T$ (columns of $H^T$) needed on a given processor in two stages. First let

\[
s_T = (Q_1^{(1)\top} \otimes I_3) t_T \quad \text{or} \quad S^T = I_3 T^T Q_1^{(1)},
\]

(2.13a,b)

where $s_T = vec(S^T)$, and then compute

\[
h_T = (I_1 \otimes Q_1^{(2)\top}) s_T \quad \text{or} \quad H^T = Q_1^{(2)\top} S^T I_1.
\]

(2.14a,b)

Alternatively, the order of the matrix multiplications in (2.12) may be reversed in which case the formulas (2.12)–(2.14) become

\[
h_T = (Q_1^{(1)\top} \otimes I_3)(I_1 \otimes Q_1^{(2)\top}) t_T \quad \text{or} \quad H^T = I_3 [Q_1^{(2)\top} T^T I_1] Q_1^{(1)},
\]

(2.15a,b)

\[
s_T = (I_1 \otimes Q_1^{(2)\top}) t_T \quad \text{or} \quad S^T = Q_1^{(2)\top} T^T I_1,
\]

(2.16a,b)

and

\[
h_T = (Q_1^{(1)\top} \otimes I_3) s_T \quad \text{or} \quad H^T = I_3 S^T Q_1^{(1)}.
\]

(2.17a,b)

For the first order of matrix multiplications in (2.12) we observe that the same load balancing to be employed in (2.8) for the first set of backsolves may be utilized in (2.13) and (2.14) so as to ensure
that only those subvectors of $h_T$ (columns of $H^T$) which are needed on a given processor (for the implementation of (2.8)) are generated on that processor; in particular, if the full matrices $T^T$ and $Q^{(2)}_1$ are available on all processors, then only those columns of $Q^{(1)}_1$ which are needed on the $i$th processor to generate the appropriate subvectors $h_T$ (columns of $H^T$) on the $i$th processor using the matrix multiplications (2.13) and (2.14), are required to be available on the $i$th processor. Similar statements apply to the reverse order of matrix multiplications in (2.15). Thus, the only interprocessor communication which the modified algorithm requires is the reshuffling of the components of $z_T$ (or, equivalently, the computation of the matrix transpose of $Z^T$) after the first set of backsolves (2.8) and before the second set of backsolves (2.9). Accordingly, the present algorithm requires a distributed matrix transpose algorithm. The price paid for interprocessor communication associated with matrix transposition is very minimal by way of comparison to the extra communication costs, higher flop count, and less than perfect load-balancing associated with using (2.10) to obtain $z$ from $h$. Further detail on the implementation and load-balancing for the algorithm (2.8), (2.9) is given in the next section.

We may observe some of the disadvantages of our previous algorithm from [12]. The algorithm (2.10), (2.11) requires the computation of $R^{(2)}^{-1}$ followed by the matrix product in (2.10) before the perfectly parallel step in (2.11) may be implemented. There are several ways in which these steps may be parallelized, but they all involve additional communication overhead and more complicated load-balancing than the algorithm of (2.8), (2.9). One approach would be to compute $R^{(2)}^{-1}$ in parallel by distributing the $p^3 + 2p/3$ flops as equally as possible across the processors by assigning columns of $R^{(2)}$ to the processors, and then combine the computation of the matrix product in (2.10) with the backsolves in (2.11) to distribute the associated flops as equally as possible across the processors by assigning rows of $R^{(2)}^{-1}$ to the processors. Another approach would be to perform a parallel matrix multiplication for $Z = HR^{(2)}^{-1}$ followed by an equidistribution of the columns of $Z$ for the backsolves in (2.11). In both cases one has a total flop count for (2.10), (2.11) which is significantly more than that for (2.8), (2.9) plus additional communication costs for generating the right hand side $z$ of (2.11) and distributing it across the processors.

### 3. The parallel algorithm

Various algorithms for Householder QR factorizations have been implemented in parallel on distributed memory computers previously [4, 7–9, 26]. We assume the availability of such an algorithm, and focus our attention on the remaining steps after the QR factorizations of $A$ and $B$ have been obtained. For the present parallel algorithm the $Q$-matrices for $A$ and $B$ were obtained in full storage mode on a serial machine and read as data on the Intel i860 computer. The steps of the present algorithm are therefore as follows:

---

1. Compute the vector $s_T$ according to $s_T = (Q^{(1)}_1 \otimes I_3) t_T$;
2. compute the vector $h_T$ according to $h_T = (I_1 \otimes Q^{(2)}_1) s_T$;
3. block back substitute to obtain the solution vector $z_T$ to the system $(I_1 \otimes R^{(2)}) z_T = h_T$;
4. form the vector $z$ from the vector $z_T$, i.e., perform a matrix transpose operation on $Z^T$;
5. block back substitute to obtain the solution vector $y$ to the system $(I_2 \otimes R^{(1)}) y = z$;
6. repermute the vector $y_T$ according to $y_T = (I_1 \otimes P_1) y_T$;
7. compute the vector $d_T$ according to $d_T = (I_1 \otimes A) q_T$;
8. form the vector $d$ from the vector $d_T$, i.e., perform a matrix transpose operation on $D^T$;
9. repermute the vector $d$ according to $d = (I_1 \otimes P_1) d$;
10. compute the residual vector $r$ according to $r = t - (I_3 \otimes B) f$.

These steps are described in detail in the following.

**Step 1.** Compute $s_T = (Q^{(1)}_1 \otimes I_3) t_T$. Let $t_T \in \mathbb{R}^{n \times n}$ and $s_T \in \mathbb{R}^{n \times q}$ be partitioned into $n$ and $q$ subvectors $t^{(i)}_T \in \mathbb{R}^{n \times n}$ and $s^{(k)}_T \in \mathbb{R}^{n \times q}$, respectively, so that $t^{(i)}_T(j) = t_T((i-1)m+j)$ and $s^{(k)}_T(j) = s_T((k-1)m+j)$ for $i = 1, \ldots, n$; $j = 1, \ldots, m$; and $k = 1, \ldots, q$. Then

$$s^{(k)}_T = T^T q^{(k)}_T \Rightarrow s^{(k)}_T(j) = \sum_{i=1}^{n} q^{(i)}_T t^{(i)}_T(j) \quad \text{for} \quad j = 1, \ldots, m; \quad k = 1, \ldots, q. \tag{3.1}$$

To compute the $j$th element in the $k$th subvector of $s_T$, we multiply the corresponding $k$th column of $Q^{(1)}_1$ by the corresponding $j$th element in each of the subvectors $t^{(i)}_T$. This computation can be performed in parallel by sending a copy of the r.h.s. vector $t$ (or $t_T$) to each processor, then sending the appropriate columns of $Q^{(1)}_1$ to each processor. If there are $N$ processors available, then each of the first $q \mod N$ processors should receive $\lfloor q/N \rfloor + 1$ columns, and the remaining processors each receive $\lfloor q/N \rfloor$ subvectors, where $\lfloor . \rfloor$ indicates the greatest integer function.

If the processors are indexed 0 through $N-1$ (as on the Intel i860), let each processor be identified by its index $id$, where $id \in \{0, 1, \ldots, N-1\}$. Let $init(id)$ and $fin(id)$ denote the initial and final columns of $Q^{(1)}_1$ to be allocated to processor $\#id$. Then the columns of $Q^{(1)}_1$ are distributed according to the following scheme (presented in pseudocode):

```plaintext
if id < (q mod N) then
  init(id) = id \cdot ([q/N] + 1) + 1,
  fin(id) = (id + 1) \cdot ([q/N] + 1)
else
  init(id) = (q mod N) \cdot ([q/N] + 1) + (id - (q mod N)) \cdot [q/N] + 1,
  fin(id) = (q mod N) \cdot ([q/N] + 1) + (id + 1 - (q mod N)) \cdot [q/N]
endif
```

Processor $\#id$ computes:

$$s^{(k)}_T(j) = \sum_{i=1}^{n} q^{(i)}_T t^{(i)}_T(j) \quad \text{for} \quad j = 1, \ldots, m; \quad k = init(id), \ldots, fin(id). \tag{3.2}$$

**Step 2.** Compute $h_T = (I_1 \otimes Q^{(2)}_1) s_T$. Let $h_T \in \mathbb{R}^{p \times n}$ be partitioned into $q$ subvectors $h^{(i)}_T \in \mathbb{R}$, so that

$$h^{(i)}_T(j) = h_T((i-1)p+j) \quad \text{for} \quad j = 1, \ldots, p; \quad i = 1, \ldots, q. \tag{3.3}$$
and we already have $s_T$ in partitioned form from the previous step. Then

$$h_T^{(i)} = Q_1^{(2)^T} s_T^{(i)} \Rightarrow h_T^{(i)}(j) = \sum_{k=1}^{m} q_k^{(2)} s_T^{(i)}(k) \quad \text{for } i = 1, \ldots, q; \ j = 1, \ldots, p. \quad (3.4)$$

This computation can be performed in parallel by sending a copy of $Q_1^{(2)}$ to each processor. Processor #id computes values for its portion of $h_T$ according to

$$h_T^{(i)} = Q_1^{(2)^T} s_T^{(i)} \quad \text{for } i = \text{init}(id), \ldots, \text{fin}(id). \quad (3.5)$$

The subvectors $s_T^{(i)}$ that are needed for this step were computed on the same processor at the previous step, so no additional communications are required.

Step 3. Solve by back substitution the block diagonal system $(I_1 \otimes R^{(2)}) z_T = h_T$. Let $z_T \in \mathbb{R}^{pq}$ be partitioned into $q$ subvectors $z_T^{(i)} \in \mathbb{R}^p$, so that

$$z_T^{(i)}(j) = z_T((i - 1)p + j) \quad \text{for } j = 1, \ldots, p; \ i = 1, \ldots, q; \quad (3.6)$$

and now we have $h_T$ in partitioned form from the previous step. Then

$$(I_1 \otimes R^{(2)}) z_T = h_T \iff R^{(2)^T} z_T^{(i)} = h_T^{(i)} \quad \text{for } i = 1, \ldots, q. \quad (3.7)$$

A copy of $R^{(2)}$ must be sent to each processor for this step. Processor #id computes values for its portion of $z_T$ by back-solving:

$$R^{(2)^T} z_T^{(i)} = h_T^{(i)} \quad \text{for } i = \text{init}(id), \ldots, \text{fin}(id). \quad (3.8)$$

Again the subvectors $h_T^{(i)}$ that are needed for this step were computed on the same processor at the previous step, so no additional communications are required.

Step 4. In order to obtain the vector $z$ from the vector $z_T$, the components of $z_T$ must be redistributed across the processors to provide the necessary subvectors $z^{(i)}$ on the appropriate processors. In other words, we must perform a matrix transpose operation on $Z_T$. Let $z \in \mathbb{R}^{pq}$ be partitioned into $p$ subvectors $z^{(i)} \in \mathbb{R}^q$, so that

$$z^{(i)}(j) = z((j - 1)q + i) = z_T^{(i)}(j) = z_T((i - 1)p + j) \quad \text{for } j = 1, \ldots, p; \ i = 1, \ldots, q. \quad (3.9)$$

The subvectors $z^{(i)}$ will be distributed to the processors according to a scheme similar (with $p$ replacing $q$) to that used in Step 1 to distribute the columns of $Q_1^{(1)}$. Now we let init1(id) and fin1(id) denote the initial and final subvectors $z^{(i)}$ to be allocated to processor #id.

The matrix transpose operation [18] uses a complete binary exchange communication scheme in which each processor makes communication with its nearest neighbors. In the first step, each processor exchanges information with the processor with binary id differing only in the first binary digit. In the second step each processor exchanges information with the processor with binary id differing only in the second binary digit, and so on. This communication scheme requires $\log_2 N$ steps, and the amount of data transmitted in each step is $2^{\log_2 N - 1}$ blocks. The size of each block to be sent from processor #id1 to processor #id2 is approximately $[\text{fin}(id2) - \text{init}(id1)] \cdot [\text{fin}(id2) - \text{init}(id2) + 1]$.

Step 5. Solve by back substitution the block diagonal system $(I_2 \otimes R^{(1)}) y = z$. Let $y \in \mathbb{R}^{pq}$ be partitioned into $p$ subvectors $y^{(i)} \in \mathbb{R}^2$, so that

$$y^{(i)}(j) = y((i - 1)q + j) \quad \text{for } i = 1, \ldots, p; \ j = 1, \ldots, q; \quad (3.10)$$
and again we have \( z \) in partitioned form from the previous step. Then
\[
(I_2 \otimes R^{(1)}) y \leftrightarrow R^{(1)} y^{(i)} = z^{(i)} \quad \text{for} \quad i = 1, \ldots, p.
\]  
(3.11)

A copy of \( R^{(1)} \) must be sent to each processor for this step. Processor \#id computes values for its portion of \( y \) by back-solving:
\[
R^{(1)} y^{(i)} = z^{(i)} \quad \text{for} \quad i = \text{init1}(id), \ldots, \text{fin1}(id).
\]  
(3.12)

Step 6. The repermutation of the vector \( y_T \) to produce the solution \( x_T \) of the original problem is obtained by
\[
x_T = (P_1 \otimes P_2) y_T.
\]  
(3.13)

Each processor repermutes only its part of the vector \( y_T \) according to \( g_T = (I_1 \otimes P_2) y_T \). Let \( g_T \in \mathbb{R}^m \) be partitioned into \( q \) subvectors \( g_T^{(i)} \in \mathbb{R}^q \), so that
\[
g_T^{(i)}(j) = g_T((i - 1)p + j) \quad \text{for} \quad i = 1, \ldots, q; \quad j = 1, \ldots, p.
\]  
(3.14)

The vector \( y_T \) is already distributed in partitioned form from the previous step. Processor \#id repermutes its part of the vector \( y_T \) according to
\[
g_T^{(i)}(j) = y_T^{(i)}(p_2(j)) \quad \text{for} \quad j = 1, \ldots, p; \quad i = \text{init1}(id), \ldots, \text{fin1}(id);
\]  
(3.15)

where \( p_2(j) \) is a pointer to the \( j \)th column of \( P_2 \).

Step 7. Compute \( d_T = (I_1 \otimes A) g_T \). Let \( d_T \in \mathbb{R}^m \) be partitioned into \( q \) subvectors \( d_T^{(i)} \in \mathbb{R}^m \), so that
\[
d_T^{(i)}(j) = d_T((i - 1)m + j) \quad \text{for} \quad i = 1, \ldots, q; \quad j = 1, \ldots, m.
\]  
(3.16)

We have \( g_T \) in partitioned form from the previous step. Then
\[
d_T = (I_1 \otimes A) g_T \leftrightarrow A g_T^{(i)} = d_T^{(i)} \quad \text{for} \quad i = 1, \ldots, q.
\]  
(3.17)

A copy of \( A \) must be sent to each processor for this step. Processor \#id computes values for its portion of \( d_T \) as follows:
\[
A g_T^{(i)} = d_T^{(i)} \quad \text{for} \quad i = \text{init1}(id), \ldots, \text{fin1}(id).
\]  
(3.18)

Step 8. In order to form the vector \( d \) from the vector \( d_T \), the components of \( d_T \) must be redistributed across the processors to provide the necessary subvectors \( d^{(i)} \) on the appropriate processors. Let \( d \in \mathbb{R}^m \) be partitioned into \( m \) subvectors \( d^{(i)} \in \mathbb{R}^q \), so that
\[
d^{(i)}(i) = d((j - 1)q + i) = d_T^{(i)}(j) = d_T((i - 1)m + j) \quad \text{for} \quad j = 1, \ldots, m; \quad i = 1, \ldots, q.
\]  
(3.19)

The subvectors \( d^{(i)} \) will be distributed to the processors according to a scheme similar (with \( m \) replacing \( q \)) to that used in step 1. Here we let \( \text{init2}(id) \) and \( \text{fin2}(id) \) denote the initial and final subvectors \( d^{(i)} \) to be allocated to processor \#id. Again we have used a complete binary exchange scheme to perform the matrix transpose operation in order to form the vector \( d \) from the vector \( d_T \).

Step 9. The repermutation of the vector \( d \) to get the vector \( f \) such that \( f = (I_3 \otimes P_1)d \) is as follows. Let \( f \in \mathbb{R}^m \) be partitioned into \( m \) subvectors \( f^{(i)} \in \mathbb{R}^q \), so that
\[
f^{(i)}(i) = f((j - 1)q + i) \quad \text{for} \quad j = 1, \ldots, m; \quad i = 1, \ldots, q.
\]  
(3.20)
where we have the vector \( d \) in partitioned form from the previous step. Then processor \#id repermutes its portion of the vector \( d \) according to

\[
F^{(j)}(i) = d^{(j)}(p_1(i)) \quad \text{for } i = 1, \ldots , m; \quad j = \text{init2}(id), \ldots , \text{fin2}(id),
\]

where \( p_1(i) \) is a pointer to the \( i \)th column of \( P_1 \).

**Step 10.** Compute \( r = t - (I_5 \otimes B)f \). Let \( r \in \mathbb{R}^{mn} \) and \( t \in \mathbb{R}^{mn} \) each be partitioned into \( m \) subvectors \( r^{(j)} \in \mathbb{R}^n \) and \( t^{(j)} \in \mathbb{R}^n \), respectively, so that

\[
r^{(j)}(k) = r((j - 1)n + k) \quad \text{and} \quad t^{(j)}(k) = t((j - 1)n + k) \quad \text{for } k = 1, \ldots , n; \quad j = 1, \ldots , m.
\]

We have \( f \) in partitioned form from the previous step. Then

\[
r = t - (I_5 \otimes B)f \Leftrightarrow r^{(j)} = t^{(j)} - B f^{(j)} \quad \text{for } j = 1, \ldots , m.
\]

A copy of matrix \( B \) must be sent to each processor for this step. Processor \#id computes

\[
r^{(j)} = t^{(j)} - B f^{(j)} \quad \text{for } j = \text{init2}(id), \ldots , \text{fin2}(id).
\]

Finally, there remains the computation of the Euclidian norm \( || \cdot ||_2 \) of the residual. Since the residual is distributed over the processors, we will compute the norm in two steps. First, each processor \#id computes the sum of the squared terms \( r^2_j \) over \( j = \text{init}(id), \ldots , \text{fin}(id) \), for its part of the residual. Second, each processor \#id > 0 sends its contribution, \( || \cdot ||_2^2 \), to processor \#0. Then, processor \#0 computes the Euclidian norm according to

\[
||r||_2^2 = \sum_{id=0}^{N-1} || \cdot ||_2^2_{id}.
\]

### 4. Computational complexities

The operation counts required to solve the least squares problem (1.3) using the QR-method consisting of (2.1) along with (2.12) and (2.8)-(2.9) are as follows:

- **Householder QR factorization of \( A \) (in factored form):** \( 2p^2(m - \frac{p}{3}) \) flops.
- **Householder QR factorization of \( B \) (in factored form):** \( 2q^2(n - \frac{q}{3}) \) flops.
- **Generation of full \( Q_2 \) matrix from Householder vectors:** \( 4(m^2 p - mp^2 + \frac{p^3}{3}) \) flops.\(^2\)
- **Generation of full \( Q_1 \) matrix from Householder vectors:** \( 4(n^2 q - nq^2 + \frac{q^3}{3}) \) flops.
- **Formation of RHS vector \( h_T \) (or \( H^T \)) in (2.12):** (using (2.13) and (2.14)) \( q[p(2m - 1) + m(2n - 1)] \) flops, or (using (2.16) and (2.17)) \( p[n(2m - 1) + q(2n - 1)] \) flops.
- **Solution of (2.8), (3.7) by backsubstitution:** \( pq^2 \) flops.
- **Solution of (2.9), (3.11) by backsubstitution:** \( q^2 p \) flops.

It follows that the total flop count for our QR-method is (using (2.13) and (2.14))

\[
p^2[q + 2(\frac{1}{3} p - m)] + q^2[p + (2(\frac{1}{3} q - n))] + 4(m^2 p + n^2 q) + 2mq(n + p) - q(m + p),
\]

\(^2\) This count, required in the current implementation, was not included in the counts reported in [12, p. 225]; in general, in the most efficient implementation, one would try to avoid explicit formation and storage of the \( Q \)-matrices.
or, when \( m = p = n = q \),

\[
\frac{14}{3} p^3 + O(p^3) \text{ flops.}
\]  

(4.2)

The part of the above least squares algorithm which was implemented on the Intel i860 computer consisted of all steps after the Householder QR factorizations were obtained, and the \( Q \)-matrices generated from them in full storage mode. These preliminary steps were performed on a serial machine using LAPACK routines DGEQPF and DORGQR, and the output for the \( Q \)'s and \( R \)'s were read as input to the parallel algorithm on the Intel i860 computer. Accordingly, the total flop count for the parallel part of the algorithm (steps 1-6) (using (2.13) and (2.14)) is

\[
q[p(2m-1) + m(2n-1)] + pq(p + q),
\]  

(4.3)

or, when \( m = n = p = q \),

\[
6p^3 - 2p^2 \text{ flops.}
\]  

(4.4)

(Here the least squares solution \( x \) is obtained from \( y \) by permutations of the components in step 6, which requires no floating point operations.) Timing results on the Intel i860 computer for the parallel part of the algorithm (steps 1-6) in the case \( m = n = p = q \) are given in Table 1 of the next section.

It remains to interface the present parallel algorithm for steps 1-6 with a parallel implementation of the Householder QR factorization. It would be highly desirable if such an interfacing would permit the right hand side vector \( h \) to be obtained using a parallel computation that would make use of the Householder QR factorizations in factored form, as this would eliminate the need and extra expense of generating the \( Q \)-matrices in full storage mode.

The steps 1 and 2 of the present implementation require that the first \( p \) columns of \( Q_2 \) be available on all processors, and that those columns of \( Q_1 \) used on a given processor in Eq. (3.2) be available on that processor. This may be expensive on storage and flop counts (for generating the columns of \( Q_1 \) and \( Q_2 \)), but has the advantage of no further interprocessor communication requirements once the appropriate columns of the \( Q \)-matrices have been passed to the processors.

It is of interest to compare the total flop count in (4.1) and (4.2) with the corresponding flop count for solving the standard normal equations by Gaussian elimination. Multiplying Eq. (1.3b) on the right by \( A \) and on the left by \( B^T \) and moving the inverses, \((A^T A)^{-1}\) and \((B^T B)^{-1}\), to the right-hand side, the solution \( X \) of the Kronecker product least squares problem may be written as

\[
X = B^T TA^{-1}
\]  

(4.5)

where

\[
A^* = (A^T A)^{-1} A^T \quad \text{and} \quad B^* = (B^T B)^{-1} B^T
\]  

(4.6)

are the Moore–Penrose pseudo-inverses of \( A \) and \( B \). The solution of the least squares problem (1.3) using (4.5) and (4.6) is widespread in the engineering literature. (In photogrammetric applications and in the geodetic sciences, where huge systems of equations are routinely encountered, Gaussian elimination on (4.6) is very commonly used and represents the workhorse for “array algebra” [28]
and its many ramifications.) The operation counts required for implementing the solution in this manner are:

Formation of $A^TA$: $p^2(2m-1)$ flops.
LU decomposition of $A^TA$: $\frac{2}{3}p^3 - \frac{1}{2}p^2 - \frac{1}{6}p$ flops.
Solution of $A^TAZ = A^T$ by Gaussian elimination: $2p^2m$ flops.\(^3\)
Formation of $B^TB$: $q^2(2n-1)$ flops.
LU decomposition of $B^TB$: $\frac{2}{3}q^3 - \frac{1}{2}q^2 - \frac{1}{6}q$ flops.
Solution of $B^TBZ = B^T$ by Gaussian elimination: $2q^2n$ flops.
Matrix product $B^+TA^+$ (multiplying $T \cdot A^+$ first): $2np(m + q) - p(q + n)$ flops.
Matrix product $B^+TA^+$ (multiplying $B^+ \cdot T$ first): $2qm(p + n) - q(m + p)$ flops.

Here we have ignored flops associated with pivot operations. It follows that the total flop count for solving the least squares problem by Gaussian elimination on the normal equations is (when the first multiplication in (4.5) is done first)

$$2p^2(2m + \frac{1}{3}p) + 2q^2(2n + \frac{1}{3}q) + 2qm(p + n) + O(p^2 + mp + q^2 + nq + qm + qp),$$

(4.7)
or, when $m = p = n = q$,

$$\frac{40}{3}p^3 + O(p^2)$$

(4.8)

The present QR-method is preferable to Gaussian elimination on numerical grounds since it enjoys stability, while the explicit formation of $A^TA$ and $B^TB$ is known to be unstable, cf. Björck [5, p. 338]. In addition, from (4.2) and (4.8), it is sometimes less expensive as well. In general, the dominant terms in (4.7) minus those in (4.1) is

$$mp(6p - 4m) + nq(6q - 4n) - pq(p + q).$$

(4.9)

Whether this is positive or negative depends on the values of $m, p, n,$ and $q$; for example, it is negative (indicating fewer flops for Gaussian Elimination) when $p < \frac{1}{3}m$ and $q < \frac{1}{3}n$.

5. Timing results

Efficiency and speed-up for parallel algorithms can be measured in several ways. Here we make use of the following standard definitions for speedup, $S_p$ [25], efficiency $E_p$ [25], and communication penalty, $C_p$ [2]:

$$S_p = \frac{\text{execution time using a single processor}}{\text{execution time using } p \text{ processors}},$$

(5.1)

$$E_p = \frac{S_p}{p},$$

(5.2)

\(^3\)This count is less than the corresponding count reported in [12, p. 225], which was based on explicit formation of $(A^TA)^{-1}$ followed by matrix multiplication with $A^T$.\end{document}
\[ C_p = \frac{\text{execution time using } p \text{ processors}}{\text{computation time using } p \text{ processors}} \tag{5.3} \]

Here

\[
\text{execution time} = \text{computation time} + \text{communication time},
\]

where, on a distributed memory computer, a reading of each of these times is available for each processor.

In Table 1 are displayed the timing data for steps 1–6 of the algorithm and the residual calculation in steps 7–10 when the \( A \) and \( B \) matrices are square of order 480×480. (This was the largest size of square \( A \) and \( B \) matrices for which the node memory, when running on one node, was not exceeded on the Intel i860 computer.) For steps 1–6 of the algorithm the only interprocessor communication occurs in the binary exchange algorithm for the transpose of \( Z^T \), that is, the generation of \( z \) from \( z_T \). We have therefore divided the transpose time into two parts, the computation time and the communication time; the computation time for the binary exchange algorithm is the time spent moving data within a node, and communication time for the binary exchange algorithm is the time spent sending data from one node to another. The total execution time, total computation time, and total communication time for steps 1–6 are given in columns 9, 10, 11 respectively. All times are in milliseconds on the Intel i860 computer. All times listed were measured on node #0.

In Table 2 are displayed the speed-up measures associated with the data in Table 1 for steps 1–6 of the algorithm, that is, for the computation of the least squares solution after the QR-factorizations of the \( A \) and \( B \) matrices have been performed.

<table>
<thead>
<tr>
<th>( N )</th>
<th>( s_T )</th>
<th>( h_T )</th>
<th>B-solves</th>
<th>Transp. comp.</th>
<th>Transp. comm.</th>
<th>B-solves</th>
<th>( g_T )</th>
<th>Exec. time</th>
<th>Comp. time</th>
<th>Speed-up</th>
<th>( S_{2p}/S_p )</th>
<th>( E_p )</th>
<th>( C_p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>146</td>
<td>104</td>
<td>85 035</td>
<td>0</td>
<td>0</td>
<td>84 558</td>
<td>236</td>
<td>470 155</td>
<td>470 155</td>
<td>0</td>
<td>317 071</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>73</td>
<td>052</td>
<td>42 515</td>
<td>274</td>
<td>331</td>
<td>42 284</td>
<td>118</td>
<td>235 686</td>
<td>235 355</td>
<td>331</td>
<td>158 810</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>36</td>
<td>525</td>
<td>21 245</td>
<td>275</td>
<td>331</td>
<td>21 140</td>
<td>59</td>
<td>118 131</td>
<td>117 800</td>
<td>331</td>
<td>79 535</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>18</td>
<td>263</td>
<td>10 609</td>
<td>207</td>
<td>250</td>
<td>10 566</td>
<td>30</td>
<td>59 203</td>
<td>58 953</td>
<td>250</td>
<td>39 823</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>9</td>
<td>131</td>
<td>5 302</td>
<td>139</td>
<td>168</td>
<td>5 284</td>
<td>14</td>
<td>29 677</td>
<td>29 509</td>
<td>168</td>
<td>19 946</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>4.56</td>
<td>6</td>
<td>2 650</td>
<td>89</td>
<td>112</td>
<td>2 642</td>
<td>7</td>
<td>14 885</td>
<td>14 773</td>
<td>112</td>
<td>10 172</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( N )</th>
<th>Exec. time</th>
<th>Comp. time</th>
<th>( S_p )</th>
<th>( S_{2p}/S_p )</th>
<th>( E_p )</th>
<th>( C_p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>470 155</td>
<td>470 155</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>235 686</td>
<td>235 355</td>
<td>1.995</td>
<td>1.995</td>
<td>0.9975</td>
<td>1.0014</td>
</tr>
<tr>
<td>4</td>
<td>118 131</td>
<td>117 800</td>
<td>3.980</td>
<td>1.995</td>
<td>0.9950</td>
<td>1.0028</td>
</tr>
<tr>
<td>8</td>
<td>59 203</td>
<td>58 953</td>
<td>7.941</td>
<td>1.995</td>
<td>0.9926</td>
<td>1.0042</td>
</tr>
<tr>
<td>16</td>
<td>29 677</td>
<td>29 509</td>
<td>15.842</td>
<td>1.995</td>
<td>0.9901</td>
<td>1.0057</td>
</tr>
<tr>
<td>32</td>
<td>14 885</td>
<td>14 773</td>
<td>31.586</td>
<td>1.99375</td>
<td>0.98706</td>
<td>1.0076</td>
</tr>
</tbody>
</table>
As is seen, the speed-up shows a slight decrease as $N$ increases; in column 5 of Table 2 we observe this more clearly in the quantity

$$\frac{S_{2p}}{S_p} = \frac{\text{execution time on } p \text{ processors}}{\text{execution time on } 2p \text{ processors}}$$

(5.4)

The nearly perfect speed-ups exhibited in Table 2 are indicative of the fact that the load balancing for the algorithm is good. To assess the effectiveness of the load balancing described in Section 3 we list in Table 3 the mean values over all processors (and associated standard deviations) of the total execution times, the total computation times, and the total communication times. The data listed are for steps 1–10, that is, the algorithm plus residual calculation. Here, total communication time includes communication times from steps 4, 8 and 10. The standard deviations, at each value of $N$ for $480 \times 480$ matrices, are only a small fraction of the means for each of the above times, indicating that all processors are doing about the same amount of work. The load balancing described in Sections 2 and 3 is therefore nearly perfect.

Another way of measuring the efficiency of a parallel algorithm has been suggested by Cleve Moler [24]: look at the scale-up properties as the total computational workload and total number of processors are simultaneously increased by a common factor. If an algorithm has perfect scale-up, then the execution time would remain constant when the total computational workload and the total number of processors were increased by the same factor. Generally, one would expect to see near-perfect scale-ups when the load balancing is good and the ratio of communication time to execution time is small. For steps 1–6 of the present algorithm, which correspond to obtaining the least squares solution (without residual), the computational workload when $A$ and $B$ are square $n \times n$ matrices is $2n^2(3n - 1)$ flops (ignoring the bookkeeping required in steps 4 and 6). If the order of $A$ and $B$ is doubled to $N = 2n$, then the total computational workload is $2N^2(3N - 1) = 4(2n^2)(2(3n) - 1)$, or approximately $8[2n^2(3n - 1)]$; that is, the computational workload is multiplied by a factor of
8 when the order of the $A$ and $B$ matrices is doubled. To examine the scale-ups of the present algorithm we should therefore increase the number of processors by a factor of 8 each time the order $n$ is doubled in order to keep the number of flops per processor approximately the same. For example, when $n = 240$, the number of flops is 82,828,800; and with $N = 2n = 480$, the number of flops becomes 663,091,200. The ratio of these numbers is 8.005563. In Table 4 are displayed the total execution times corresponding to scale-ups from $n = 200$ to $N = 400$, and from $n = 240$ to $N = 480$. In each case eight times as many processors are used for the larger matrices than are used for the smaller matrices. The total execution times listed in columns 2 and 4 (and columns 6 and 8) remain roughly the same, indicating that the algorithm has near-perfect scale-up.

6. Conclusions

The parallel algorithm for the Kronecker product least squares problem combines the desirable stability properties of the QR-approach with a computational scheme that requires minimal computer memory. Since applications typically involve very large $A$ and $B$ matrices this is an essential consideration. The algorithm exhibits near perfect load-balancing and, as a consequence, very good computational efficiency on distributed memory computers.

7. Acknowledgements

The authors thank James Ortega for suggesting this approach to the problem. We thank both James Ortega and Vance Faber for calling attention to the commutativity property of the Kronecker product and its utility for the present algorithm. We also thank our colleagues, Gary Howell and William Shoaff, who participated in a Parallel Computing Seminar at Florida Institute of Technology, and offered useful help during the time when this software was being developed. We express our appreciation to the Advanced Computing Laboratory of Oak Ridge National Laboratory for providing ample time on their Intel i860 computer for testing and development of our parallel codes, and to Judy Green and Tom Dunigan for their help in using ORNL computing facilities. Finally, we thank the referee for a very critical reading of the manuscript which resulted in a few corrections and some improvements in the exposition.

References


