A Parallel Ordering for the Block Jacobi Method on a Hypercube Architecture*

Christian Bischof

TR 86-740
March 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

* This research was partially supported by ONR contract N 0001483-K-0640

A Parallel Ordering for the Block Jacobi Method on a Hypercube Architecture[1]

Christian Bischof

March 17, 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract

Jacobi methods for computing the singular value decomposition are ideally suited for multiprocessor environments since they contain a great deal of inherent parallelism. We give a parallel ordering for the block Jacobi Method that allows us to take full advantage of the nearest-neighbour topology of the hypercube. It is based on recursively embedding smaller rings into the hypercube using a Gray code labelling of processor nodes. This scheme is optimal in that it entails the lowest communication overhead possible.

---

[1] This research was partially supported by ONR contract N 0001483-K-0640.

§1. Introduction

The singular value decomposition (SVD) of a matrix $A \in \mathbb{R}^{m \times n}$ $(m \geq n)$ is a very versatile decomposition and perfectly stable to compute. See Golub and Van Loan (1983). In the SVD we seek orthogonal matrices $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ such that

$$U^T A\, V = diag(\sigma_1, \ldots, \sigma_n)$$

The σ_i are called the singular values of A. It is normally assumed that $\sigma_1 \geq \ldots \geq \sigma_n \geq 0$, but we will not insist on this normalisation.

To set the stage for the block Jacobi algorithm, consider A as a $k \times k$ block matrix with each block $A_{ij} \in \mathbb{R}^{m/k \times n/k}$. Block Jacobi methods proceed by making A increasingly more block-diagonal. To measure how close A is to the desired block diagonal form, we define OFF(A) as the Frobenius norm of the off-diagonal blocks, i.e.

$$\mathrm{OFF}(A) \equiv \mathrm{sqrt}\Big(\sum_{i \neq j} \|A_{ij}\|_F^2\Big)$$

We consider A to be block-diagonal if OFF(A) is in the order of machine precision. Once A is block diagonal, it is straightforward to compute the SVD of A by computing the SVD of each diagonal block.

To decrease OFF(A) we repeatedly compute the SVD of two-by-two block subproblems S_{ij} of the form

$$S_{ij} \equiv \begin{bmatrix} A_{ii} & A_{ij} \\ A_{ji} & A_{jj} \end{bmatrix}, \text{ where } i<j.$$

Now let

$$W_{ij} \equiv \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix} \text{ and } Z_{ij} \equiv \begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix}$$

be such that

$$W_{ij}^T S_{ij} Z_{ij} = \begin{bmatrix} D_{ii} & 0 \\ 0 & D_{jj} \end{bmatrix} \equiv \Sigma_{ij}$$

where D_{ii} and D_{jj} are diagonal $m/k \times n/k$ matrices. Further let $J(W,i,j) \equiv (X_{rs})$ be the $k \times k$ block matrix that is the identity everywhere except $X_{ii} = W_{11}$, $X_{ij} = W_{12}$, $X_{ji} = W_{21}$ and $X_{jj} = W_{22}$. $J(Z,i,j)$ is defined in an analogous manner. Then it is not hard to see that

$$\text{(1.1)} \qquad \mathrm{OFF}(J(W,i,j)^T A\, J(Z,i,j))^2 = \mathrm{OFF}(A)^2 - \| A_{ij} \|_F^2 - \| A_{ji} \|_F^2$$

So by solving a judiciously chosen sequence of two-by-two block subproblems we will be able to reduce A to block diagonal form.

In §2 we give both the serial and parallel version of the block Jacobi algorithm. In §3 we discuss a scheme for zeroing the off-diagonal blocks that allows for maximal concurrency with minimal communication overhead on a hypercube architecture. We summarize our results in §4.

§2. The Block Jacobi Algorithm

The general block Jacobi algorithm we will present has been studied by Van Loan (1985). The block Jacobi algorithm proceeds in *block sweeps*. Let

$$\Gamma \equiv \{ (i,j) \mid 1 \le i < j \le k \}$$

be the set of off-diagonal index pairs. During each block sweep the subproblem S_{ij} is solved exactly once for every index pair $(i,j) \in \Gamma$. By then updating A with $J(W,i,j)$ and $J(Z,i,j)$ we steadily decrease $\mathrm{OFF}(A)$ according to (1.1). Note that off-diagonal blocks that had been zeroed in one subproblem will be filled in later when we update A with $J(W,i,j)$ and $J(Z,i,j)$ for another subproblem.

To guard against pathological cases and ensure convergence of the algorithm it is necessary to incorporate a threshold criterion into the algorithm. We skip solving for the S_{ij} subproblem if

$$\mu_{ij} \equiv \mathrm{sqrt}(\| A_{ij} \|_F^2 + \| A_{ji} \|_F^2)$$

is small according to a subproblem threshold τ. To determine the sequence in which we solve the subproblems , let

$$\Phi_s \equiv ((i_1^s , j_1^s) , \ldots , (i_r^s , j_r^s)), \quad r = \frac{k\,(k-1)}{2}$$

be an ordering of Γ. Φ_s is called the *block sweep ordering* for sweep s and will determine the order in which we solve the subproblems S_{ij} in block sweep s. So with $A = [A_1 , \ldots , A_k]$, $U = [U_1 , \ldots , U_k]$, $V = [V_1,\ldots,V_k]$ and I_m the $m \times m$ identity we obtain Algorithm 2.1 . Van Loan (1985) has proved the convergence of this algorithm and in practice it converges quadratically.

Algorithm 2.1.

Given a $k \times k$ block matrix A, $\varepsilon > 0$, a sequence $\Phi \equiv (\phi_1, \phi_2, \ldots)$ of block sweep orderings and a subproblem threshold $\tau \le \varepsilon \| A \|_F / k$, the following algorithm computes orthogonal U and V such that $\mathrm{OFF}(U^T A V) \le \varepsilon \| A \|_F$. The original A gets overwritten with $U^T A V$.

$U \leftarrow I_m$; $V \leftarrow I_n$; $s \leftarrow 1$; $r \leftarrow k(k-1)/2$;
<u>while</u> $\mathrm{OFF}(A) > \varepsilon \| A \|_F$ <u>do</u>
    <u>for</u> $l = 1$ <u>to</u> r <u>do</u>
      $(i,j) \leftarrow (i_l^{(s)}, j_l^{(s)})$
      <u>if</u> $\mu_{ij} \ge \tau$ <u>then</u>
        Solve subproblem S_{ij} by computing W_{ij}, V_{ij} and Σ_{ij} such that
          $W_{ij}^T S_{ij} Z_{ij} = \Sigma_{ij}$
        Update
          $A \leftarrow J(W,i,j)^T A\, J(Z,i,j)$; $U \leftarrow U\, J(W,i,j)$; $V \leftarrow V\, J(Z,i,j)$
        by computing
          $[A_i, A_j] \leftarrow [A_i, A_j] Z_{ij}$
          $[U_i, U_j] \leftarrow [U_i, U_j] W_{ij}$
          $[V_i, V_j] \leftarrow [V_i, V_j] Z_{ij}$
          <u>for</u> $q = 1$ <u>to</u> k <u>do</u>

$$\begin{bmatrix} A_{i,q} \\ A_{j,q} \end{bmatrix} \leftarrow W_{ij}^T \begin{bmatrix} A_{i,q} \\ A_{j,q} \end{bmatrix} \qquad (2.1)$$

          <u>end for</u>
      <u>end if</u>
    <u>end for</u>
    $s \leftarrow s + 1$
<u>end while</u>

As far as the convergence of Algorithm 2.1 is concerned, a block sweep ordering can be *any* permutation of Γ and the orderings for different sweeps can be different. We will exploit this freedom in focussing our attention on orderings that allow the solution of several subproblems in parallel.

Now suppose A is a 8 by 8 block matrix and we have 4 processors $P_1, \ldots, P_4$ where P_i contains block columns $2i-1$ and $2i$ of A, U and V. Then we can solve subproblems S_{12}, S_{34}, S_{56} and S_{78} concurrently. The updates

$$\begin{aligned} [A_i, A_j] &\leftarrow [A_i, A_j] Z_{ij} \\ [U_i, U_j] &\leftarrow [U_i, U_j] W_{ij} \qquad (i,j) \in \{(1,2),(3,4),(5,6),(7,8)\} \\ [V_i, V_j] &\leftarrow [V_i, V_j] Z_{ij} \end{aligned}$$

can also be performed concurrently. Only for the update (2.1) each processor P_i has to broadcast its $W_{2i-1,2i}$ to every other processor since rows $2i-1$ and $2i$ are distributed over all the other processors.

In general let $\Omega = ((i_1, j_1), \ldots, (i_p, j_p))$ be an ordered subset of Γ. The p subproblems corresponding to Ω are called *nonconflicting* if all first indices are mutually distinct and all second indices are mutually distinct, i.e.

$$(2.2) \qquad \forall\, 1 \le l_1, l_2 \le p,\ l_1 \ne l_2 :\ i_{l_1} \ne i_{l_2} \text{ and } j_{l_1} \ne j_{l_2}$$

Given p processors, p nonconflicting subproblems can be solved concurrently. Of course we can have at most $k/2$ nonconflicting subproblems. An index set Ω corresponding to $k/2$ nonconflicting subproblems is called a *rotation set*. From now on we will only deal with rotation sets and to make the notation of the following algorithms easier, we do not require $i_l < j_l$ any more. So a rotation set is an ordered set $\Omega = ((i_1, j_1), \ldots, (i_p, j_p))$ of tuples such that

$$\{ i_1, j_1, \ldots, i_{k/2}, j_{k/2} \} = \{ 1, \ldots, k \}.$$

Since we are interested in exploiting fully the parallelism inherent in the problem let $p \equiv k/2$ be the number of available processors. Then we want block sweep orderings ϕ_s such that in every sweep s the first p index pairs of ϕ_s form a rotation set $\Omega_1{}^s$, the second p index pairs form a rotation set $\Omega_2{}^s$ up to the $(k-1)$st p index which form roation set $\Omega_{k-1}{}^s$. Such an ordering ϕ_s is called a *parallel block sweep ordering* and we write

$$\phi_s = (\Omega_1^s, \ldots, \Omega_{k-1}^s)$$

Given a parallel block sweep ordering we can compute all $k(k-1)/2$ subproblems on $k/2$ processors in $k-1$ time steps. Note however that after each processor has solved its subproblem the processors have to exchange block columns of A, U and V between each other to make sure that each processor has those two block columns that it needs to solve the next subproblem assigned to it.

This leads to Algorithm 2.2 which is the distributed version of Algorithm 2.1 based on a series of parallel orderings. Here $E_i{}^n$ is the ith block column of the $n \times n$ identity I_n, i.e.

$$I_n = [E_1^n, \ldots, E_k^n],\ E_i^n \in \mathbb{R}^{n \times n/k}$$

We further assume that the first rotation set in the first sweep

$$\Omega_1^1 = ((1,2),(3,4), \ldots, (k-1,k))$$

and that block columns A_{2t-1} and A_{2t} have already been downloaded into processor t, $t \in \{1,..,p\}$. How the subproblems are solved has quite an impact on the convergence speed of the algorithm and it is an open problem what is the best way to solve them. For a discussion of these issues, the reader is referred to Bischof and Van Loan (1986).

processor t

```
[ U_{2t-1} , U_{2t} ] ← [ E'''_{2t-1} , E'''_{2t} ]; [ V_{2t-1} , V_{2t} ] ← [ E''_{2t-1} , E''_{2t} ] ; s ← 1;
while OFF(A) > ε ||A||_F do
    for l = 1 to k − 1 do
        If Ω^s_l = ((i_1,j_1) , ... , (i_p,j_p)) is the lth rotation set in the sth sweep,
        then processor t solves the subproblem corresponding to (i_t^s, j_t^s )
        (i,j) ← (i_t^s, j_t^s )
        if μ_ij ≥ τ then
            Solve subproblem S_ij by computing W_ij, V_ij and Σ_ij such that
                W_ij^T S_ij Z_ij = Σ_ij
        else
            W_ij ← I_{m/k} ; Z_ij ← I_{n/k}
        end if
        Update the block columns; they reside in this processor
            [ A_i , A_j ] ← [ A_i ,A_j ] Z_ij
            [ U_i , U_j ] ← [ U_i ,U_j ] W_ij                                  (2.2)
            [ V_i , V_j ] ← [ V_i ,V_j ] Z_ij
        Update the block rows of A. They are distributed over all processors.
            Broadcast W_ij to all other processors.
            for all (x,y) ∈ {(i_1,j_1) , ... , (i_p,j_p)} do
                receive W_{x,y}
                update
                [ A_{x,i}  A_{x,j} ]              [ A_{x,i}  A_{x,j} ]
                [ A_{y,i}  A_{y,j} ]  ← W_{x,y}^T [ A_{y,i}  A_{y,j} ]
            end forall
        Exchange block columns with other processors to set up each processor
        for the next rotation set.
            Determine next rotation set.
            if l < k − 1 then /* use next rotation set in sweep s */
                let Ω^s_{l+1} = ((i_1,j_1) , ... , (i_p,j_p))
            else /* use next rotation set in sweep s + 1 */
                let Ω^{s+1}_1 = ((i_1,j_1) , ... , (i_p,j_p))
            end if
            Send out the current block column sets unless they are
            further needed in this processor.
            if (( i_q = i or j_q = i ) and q ≠ t ) then
                send [ A_i , V_i , U_i ] to processor q
            end if
            if ( ( i_q = j or j_q = j ) and q ≠ t ) then
                send [ A_j , V_j , U_j ] to processor q
            end if
            Receive the block column sets needed from other processors.
            if ( i_t ≠ i and i_t ≠ j ) then
                receive [ A_{i_t} , V_{i_t} , U_{i_t} ]
            end if
            if ( j_t ≠ i and j_t ≠ j ) then
                receive [ A_{j_t} , V_{j_t} , U_{j_t} ]
            end if
    end for
    s ← s + 1
end while
```

Algorithm 2.2 (to be executed in processor t, $t \in \{1,..,p\}$)

Given a $k \times k$ block matrix $A, \varepsilon > 0$, a sequence $\Phi \equiv (\phi_1, \phi_2, \ldots)$ of parallel block sweep orderings and a subproblem threshold $\tau \leq \varepsilon \| A \|_F / k$, the previous algorithm computes orthogonal U and V such that $\mathrm{OFF}(U^T A V) \leq \varepsilon \| A \|_F$. The original A gets overwritten with $U^T A V$.

Of course we would like to keep the communication overhead of the algorithm to a minimum and so it would be desirable if block columns of A, U and V would be exchanged only between adjacent processors. The communication time would be further reduced if each processor would exchange *only one* set of block columns between successive rotation sets. Each processor has to exchange at least one set of block columns between rotation sets as each subproblem is solved exactly once in a sweep. If we could achieve these goals the algorithm would be optimal with respect to its communication overhead.

It depends on the topology of the system whether it is possible to generate such an optimal ordering. Van Loan (1985) has suggested a parallel ordering for a ring topology where each processor communicates only with its nearest neighbours but every processor except two exchange the two sets of block columns between successive rotation sets. We will now show that it is possible, however, to generate such an optimal ordering for a hypercube topology.

§3. The Parallel Ordering for the Hypercube

The hypercube is an example of an architecture where a large number of processors are connected to one another according to a fixed pattern. The system does not have any shared memory and all communication and synchronisation is achieved through message passing. A d-dimensional hypercube contains 2^d processors. If $b(i) = b_1 \ldots b_d$ is the binary label of processor i then processor i is connected to processor j if and only if $b(i)$ and $b(j)$ differ in exactly one bit. So a hypercube of dimension d can be obtained by connecting corresponding processors of two $(d-1)$-dimensional hypercubes.

In our algorithm we will have to embed rings in the hypercube and so we need a processor numbering such that processors adjacent in the ring will be adjacent in the hypercube. The *binary reflected Gray code* fulfills this specification (see Johnson (1985) and Saad and Schultz (1985)). To define it, let $G(d)$ be a 2^d-vector whose ith entry g_i^d is the encoding of i $(i = 0,...,2^d - 1)$ in the Gray code for 2^d numbers. Then

$$G(1) \equiv \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \; G(2) \equiv \begin{bmatrix} 00 \\ 01 \\ 11 \\ 10 \end{bmatrix}, \; G(3) \equiv \begin{bmatrix} 000 \\ 001 \\ 011 \\ 010 \\ 110 \\ 111 \\ 101 \\ 100 \end{bmatrix} \text{ and in general } G(d+1) \equiv \begin{bmatrix} g_0^d\,0 \\ g_0^d\,1 \\ g_1^d\,1 \\ g_1^d\,0 \\ g_2^d\,0 \\ g_2^d\,1 \\ \cdot \\ \cdot \\ \cdot \\ g_{2^d-2}^d\,0 \\ g_{2^d-2}^d\,1 \\ g_{2^d-1}^d\,1 \\ g_{2^d-1}^d\,0 \end{bmatrix} \qquad (3.1)$$

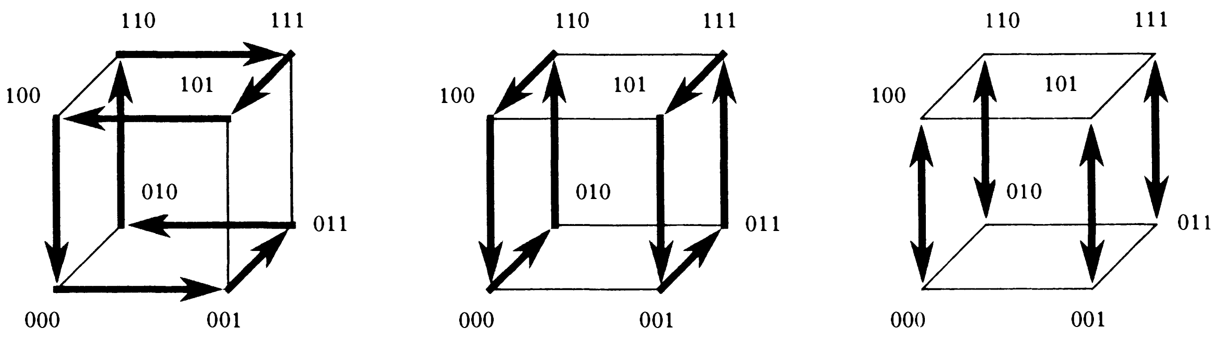
So if i and j are successive numbers (modulo 2^d) then g_i^d and g_j^d differ in exactly one bit. Now let

$$S_1 \equiv \{ g_i^d \mid i = 0 + 4l \text{ or } i = 3 + 4l, l \in \{0,1,2,...,2^{d-2} - 1\}\}$$

and (3.2)

$$S_2 \equiv \{ g_i^d \mid i = 2 + 4l \text{ or } i = 3 + 4l, l \in \{0,1,2,...,2^{d-2} - 1\}\}$$

From (3.1) it is easy to see that processors with numbers in S_1 and processors with numbers in S_2 lie on two smaller rings of size 2^{d-1} since the last bit in each group is identical and the first $d-1$ bits are numbered according to $G(d-1)$. If we now just look at the first $d-1$ bits in S_1 or S_2, we can repeat this procedure and embed the processors within each group onto two rings of size 2^{d-2}. Continuing this argument recursively, we can altogether embed 2^{d-l} rings of length 2^l in the hypercube, $l = 0,1,...,d$. Figure 1 shows the embedding of the rings on a 3-cube.



a) one ring of size 8   b) two rings of size 4   c) four rings of size 2

Figure 1: Embedding rings in a 3-cube

From now on assume that the block dimension k of A is a power of two, i.e. $k = 2^{d-1}$ and that we have $p = 2^d$ processors in a d-dimensional hypercube available. Further let the processors be labelled according to the Gray coding for a d-dimensional hypercube. So processor P_{i+1} has binary label $g_i{}^d$, $i = 0,..,2^d - 1$.

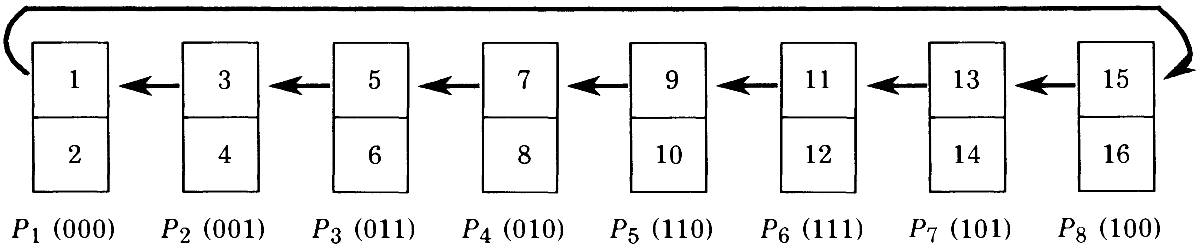
To generate the parallel ordering for the hypercube, we embed successively smaller rings in the hypercube. To that end we recursively exploit the fact that if processors $P_1, \ldots, P_{2^q}$ lie on a ring of size 2^q, then according to (3.2) processors

$$P_1, P_4, P_5, P_8, \ldots, P_{1+4l}, P_{4+4l}, \ldots, P_{2^q-3}, P_{2^q}$$

and $\qquad l = 0, \ldots, 2^{q-2} - 1$

$$P_2, P_3, P_7, P_8, \ldots, P_{2+4l}, P_{3+4l}, \ldots, P_{2^q-2}, P_{2^q-1}$$

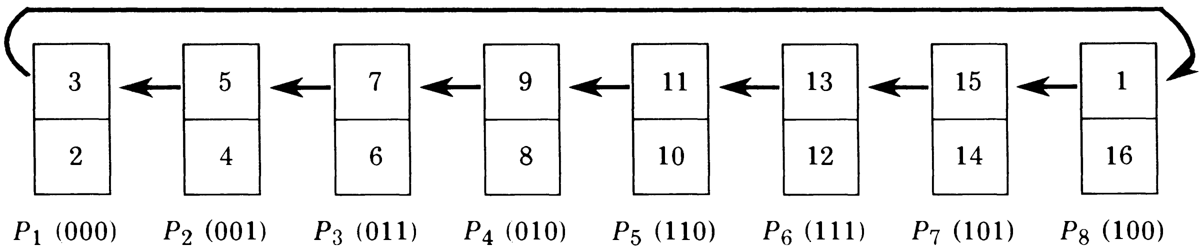
lie on two smaller rings of size 2^{q-1} each. The idea is demonstrated using an example on a 3-cube, going through one sweep for a 16×16 block matrix distributed over 8 processors. Assuming that processor P_i contains block column sets $2i-1$ and $2i$, $i = 1,..,8$, we have the following initial set-up. The binary number in parentheses is the Gray code label for the corresponding processor.



So the first rotation set would be

((1,2), (3,4), (5,6), (7,8), (9,10), (11,12), (13,14), (15,16)).

The next rotation set is generated by using the 8-ring (see Figure 1a) and sending the "upper" block column set to the left neighbour and receiving a new one from the right neighbour as indicated by the arrows. So we obtain



and the next rotation set is

((3,2), (5,4), (7,6), (9,8), (11,10), (13,12), (15,14), (1,16)).

Using the same idea of just moving the upper column again, we can now generate rotation sets

((5,2), (7,4), (9,6), (11,8), (13,10), (15,12), (1,14), (3,16))

((7,2), (9,4), (11,6), (13,8), (15,10), (1,12), (3,14), (5,16))

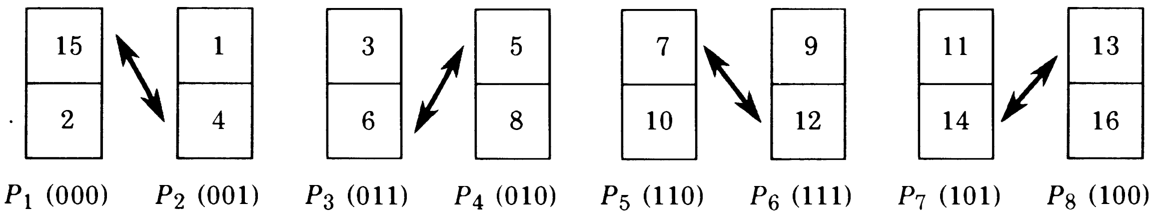
((9,2), (11,4), (13,6), (15,8), (1,10), (3,12), (5,14), (7,16))

((11,2), (13,4), (15,6), (1,8), (3,10), (5,12), (7,14), (9,16))

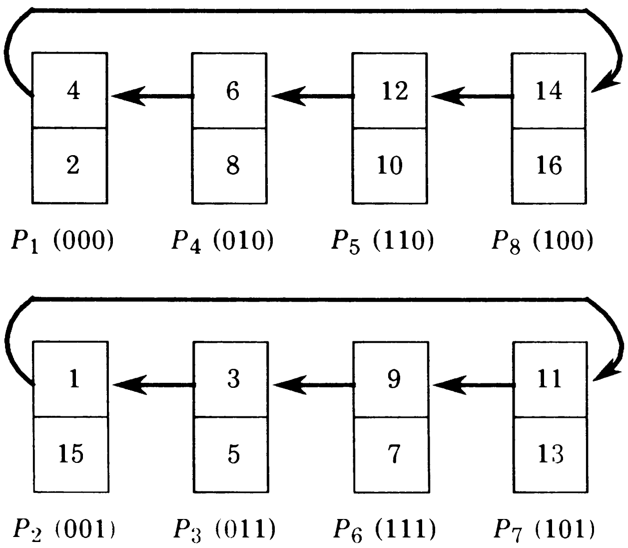
((13,2), (15,4), (1,6), (3,8), (5,10), (7,12), (9,14), (11,16))

((15,2), (1,4), (3,6), (5,8), (7,10), (9,12), (11,14), (13,16))

and then the configuration is



Now every subproblem involving an "upper" and "lower" block column set has been solved and continuing the way we did so far would just reproduce the first rotation set. So to generate the next rotation set, we split the ring of 8 processors into two smaller rings consisting of processors { 1,4,5,8 } and { 2,3,6,7 }, respectively (see Figure 1b). The block column sets are exchanged in a zig-zag manner as indicated by the arrows. The configuration we have now is



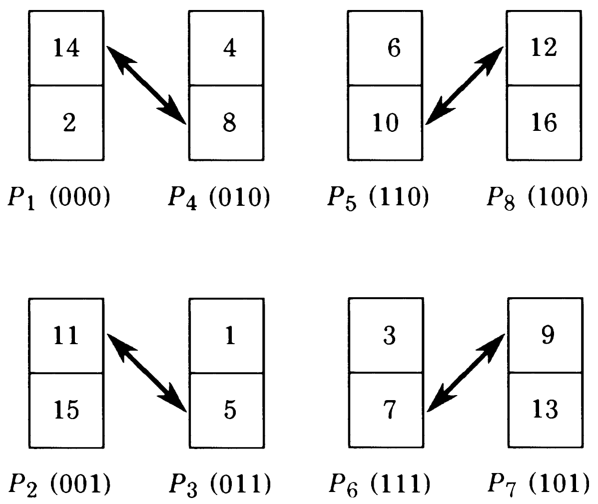
with the corresponding rotation set

((4,2), (1,15), (3,5), (6,8), (12,10), (9,7), (11,13), (14,16)).

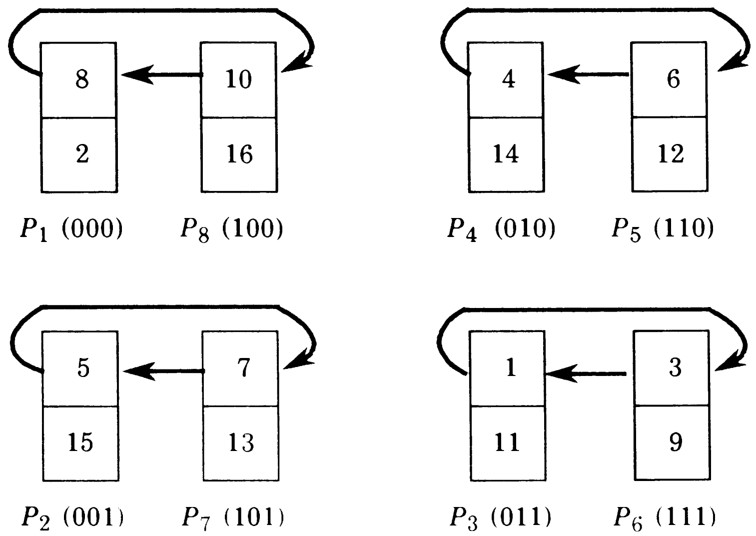
Note that the first ring now contains all the previously "lower" block columns, whereas the second ring contains all the previously "upper" columns. The next 3 rotation sets are now again generated by sending a block column to the left neighbour and receiving one from the right neighbour - independently in each 4-ring. They are

$$((6,2), (3,15), (9,5), (12,8), (14,10), (11,7), (1,13), (4,16))$$
$$((12,2), (9,15), (11,5), (14,8), (4,10), (1,7), (3,13), (6,16))$$
$$((14,2), (11,15), (1,5), (4,8), (6,10), (3,7), (9,13), (12,16))$$

Now the configuration is



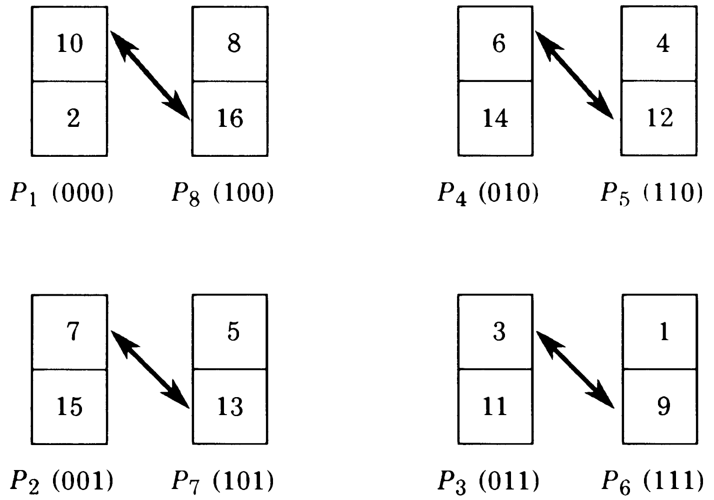
and splitting each four-ring into two rings of two processors (see Figure 1c) we obtain



with rotation set

((8,2), (5,15), (1,11), (4,14), (6,12), (3,9), (7,13), (10,16)).

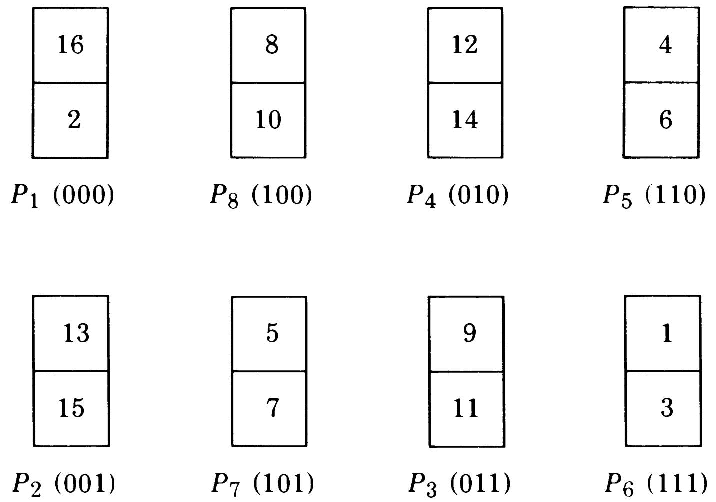
We then use the small rings to obtain



with rotation set

((10,2), (7,15), (3,11), (6,14), (4,12), (1,9), (5, 13), (8,16)).

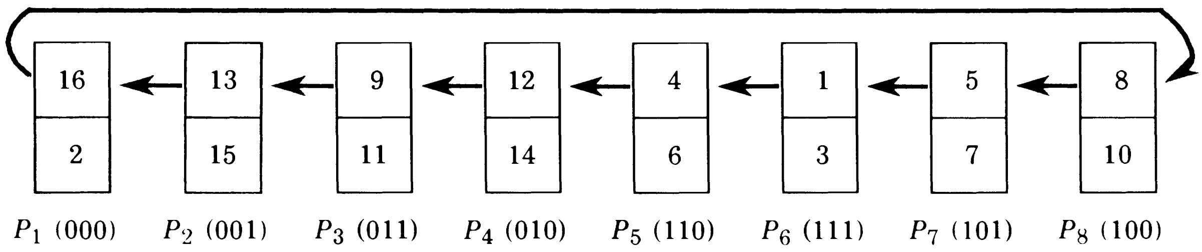
and we split one more time to obtain 8 trivial rings of one processor each



with rotation set

((16,2), (13,15), (9,11), (12,14), (4,6), (1,3), (5,7), (8, 10)).

At this point we are finished with one sweep and to generate the first rotation set for the next sweep we look at the processors again as members in the biggest ring and send the "upper" column to the left neighbour and receive a new one from the right neighbour.



So the first rotation set in the next sweep would be

$$((13,2), (9,15), (12,11), (4,14), (1,6), (5,3), (8,7), (16,10))$$

and from here on we would continue as described so far.

This example shows that this method for generating the subproblems in a sweep conforms to our definition of an optimal parallel ordering: We generate $k-1$ rotation pairs and every processor exchanges only one set of block columns between each rotation set. Note that the block sweep orderings generated by this method differ from sweep to sweep.

We now introduce the block Jacobi algorithm for the hypercube that relies on this particular ordering. A processor has to be able to identify indices of incoming messages so we assume that the message $[\, A_i\,,\, U_i\,,\, V_i\,]$ is tagged with i and W_{ij} is tagged with i and j. The W_{ij}'s are relayed to every processor by forwarding them around the 2^d-ring and they can be used to piggyback information allowing each processor to compute OFF(original A) and update it thereafter. For more detail on this see Bischof and Van Loan (1986).

Algorithm 3.1. (to be executed in processor t, $t \in \{1, .., p\}$)

Given a $k \times k$ matrix A ($k = 2^{d+1}$), $\varepsilon > 0$, a subproblem threshold $\iota \leq \varepsilon \, \| A \|_F / k$, the following algorithm computes orthogonal U and V such that OFF($U^T A V$) $\leq \varepsilon \; \| A \|_F$ on a d-dimensional hypercube. The original A gets overwritten with $U^T A V$.

processor t

Let $b_1 \dots b_d = g_{t-1}{}^d$ be the binary label of processor t.

if $t = 1$ then $LNEIB \leftarrow 2^d$ else $LNEIB \leftarrow t-1$ end if

if $t = 2^d$ then $RNEIB \leftarrow 1$ else $RNEIB \leftarrow t+1$ end if

Let P_{LNEIB} (P_{RNEIB}) be the processor i with binary label $g_{LNEIB-1}{}^d$ ($g_{RNEIB-1}{}^d$).

It is the left (right) neighbour of P_t in the 2^d-ring

$[\, U_{2t-1}, U_{2t}] \leftarrow [\, E'''_{2t-1}, E'''_{2t}]$; $[\, V_{2t-1}, V_{2t}] \leftarrow [\, E''_{2t-1}, E''_{2t}]$;

while $\mathrm{OFF}(A) > \varepsilon \, \| A \|_F$ do

  for $l = d$ to 0 do

    Let q be such that $g_{q-1}{}^l = b_1, \dots, b_l$.

    This processor is now considered the q-th processor in the 2^l-ring.

    if $q = 1$ then $lneib \leftarrow 2^l$ else $lneib \leftarrow q-1$ end if

    if $q = 2^l$ then $rneib \leftarrow 1$ else $rneib \leftarrow q+1$ end if

    Let P_{lneib} (P_{rneib}) be the processor with binary label $g_{lneib-1}{}^l\, b_{l+1} \dots b_d$ ($g_{rneib-1}{}^l\, b_{l+1} \dots b_d$). It is the left (right) neighbour of P_t in the 2^l-ring.

    Let $[\, A_i, V_i, U_i]$ and $[\, A_j, V_j, U_j]$ be the block column sets currently in this processor.

    Let B_i (B_j) be an abbreviation for $[\, A_i, V_i, U_i]$ ($[\, A_j, V_j, U_j]$).

    $ctr1 \leftarrow 1$;

    do 2^l times

      Solve the subproblem in this processor and update block columns according to (2.2).

      $(x,y) \leftarrow (i,j)$; $ctr2 \leftarrow 1$;

      Update the block rows of A. They are distributed over all processors.

      do 2^d times

        if($ctr2 \neq 1$) then

          send $W_{x,y}$ to P_{LNEIB} ; receive new $W_{x,y}$ from P_{RNEIB} ;

        end if

$$\begin{bmatrix} A_{x,i} & A_{x,j} \\ A_{y,i} & A_{y,j} \end{bmatrix} \leftarrow W_{x,y}{}^T \begin{bmatrix} A_{x,i} & A_{x,j} \\ A_{y,i} & A_{y,j} \end{bmatrix}$$

        $ctr2 \leftarrow ctr2 + 1$;

      end do

      Exchange block columns with other processors to set up each processor for the next rotation set.

      if ($ctr1 < 2^l$) then /* use the 2^l-ring */

        send B_i to P_{lneib} ; receive new B_i from P_{rneib} .

      else if $l > 0$ then /* split the ring exchanging columns in a zig-zag manner */

        $temp \leftarrow (q-1) \bmod 4$;

        case $temp$ of

          0 : send B_i to P_{rneib} ; receive new B_i from P_{rneib} ;

          1 : send B_j to P_{lneib} ; receive new B_j from P_{lneib} ;

          2 : send B_j to P_{rneib} ; receive new B_j from P_{rneib} ;

          3 : send B_i to P_{lneib} ; receive new B_i from P_{lneib} ;

        end case

      else /* end of a sweep. Use the 2^d- ring */

        send B_i to P_{LNEIB} ; receive new B_i from P_{RNEIB} ;

      end if

    end do

  end for

end while

Using a modified version of the Chan-SVD Algorithm for solving the subproblems (see Bischof and Van Loan (1986)) each sweep requires

$$\frac{1}{p}\,(2m^3 + 2n^3 + 2mn^2 + m^2n) + \frac{1}{p^2}\,(8mn^2 + 9\frac{1}{2}n^3 + 7m^2n)$$

(classical) flops on each processor. The time each processor spends on communication is

$$4\gamma p^2 + 2t\,(2m^2 + n^2 + mn)$$

where γ is the overhead for every send and receive and t is the time it takes to transfer one word. Of course counting flops is a somewhat skewed measure of performance in the parallel setting, but nonetheless these figures demonstrate that for typical values of m, n, p and t, the communication overhead is very small. This algorithm does not sort the singular values but if that is required, a variánt of bitonic sort can be used to sort the singular values in time that is negligible compared to the overall cost of the algorithm (see Bilardi and Nicolau (1986) and Johnson (1984)).

Note that the particular ordering I have suggested is only one of many that can be generated by recursively embedding smaller rings. We could as well go from smaller rings to bigger rings or communicate in different directions along the different rings. All these orderings would have the same communication complexity and there is no reason to believe that any particular of them would result in faster convergence of the algorithm than any other.

§4. Conclusions

We developed a general framework for block Jacobi methods on parallel processors and then suggested a parallel ordering for a hypercube topology. This ordering is generated by recursively embedding smaller rings into the hypercube using a Gray code labelling of the processor nodes. It is optimal in the sense that it only relies on nearest-neighbour communication and entails the lowest communication overhead possible.

Acknowledgements

I wish to thank my advisor Charles Van Loan for the many stimulating discussions about Jacobi methods in general and parallel ones in particular.

References

C. Bischof and C. Van Loan (1986), "The Block Jacobi Method on a Ring of Processors", to appear

G. Bilardi and A. Nicolau(1986), "Bitonic Sorting with O(*N*log*N*) comparisons", to appear in Proceedings of the 20th Annual Conference on Information Sciences and Systems, Princeton, NJ, March 1986

G. H. Golub and C. Van Loan (1983), *Matrix Computations*, John Hopkins University Press, Baltimore, MD

S. L. Johnson (1984), "Combining Parallel and Sequential Sorting on a Boolean *n*-Cube", Proceedings of the International Conference on Parallel Processing, 1984, Shanty Creek, MI

S. L. Johnson (1985), "Communication Efficient Basic Linear Algebra Computations on a Hypercube Architecture", Research Report YALEU/DCS/RR-361, September 1985

Y. Saad and M. H. Schultz (1985), "Topological Properties of Hypercubes", Research Report YALEU/DCS/RR-389, June 1985

C. Van Loan (1985), "The Block Jacobi Method for Computing the Singular Value Decomposition", Cornell University, Dept. of Computer Science TR-680, June 1985