Matrix Multiplication And Data Routing Using A Partitioned Optical Passive Stars Network

Sartaj Sahni
sahni@cise.ufl.edu
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611

Abstract

We develop optimal or near optimal algorithms to multiply matrices and perform commonly occurring data permutations and BPC permutations on multiprocessor computers interconnected by a partitioned optical passive stars network.

Keywords: Partitioned optical passive stars network, matrix multiplication, data routing, BPC permutations.

1 Introduction

The partitioned optical passive stars network (POPS) was proposed in [2, 5, 6, 8] as an optical interconnection network for a multiprocessor computer. The POPS network uses multiple optical passive star (OPS) couplers to construct a flexible interconnection topology. Each OPS (Figure 1) coupler can receive an optical signal from any one of its source nodes and broadcast the received signal to all of its destination nodes. The time needed to perform this receive and broadcast is referred to as a *slot*.

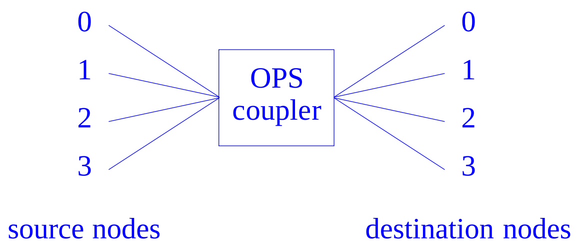

Figure 1: An optical passive star coupler with 4 source and 4 destination nodes

Although a single OPS can be used to interconnect n processors (in this case the n processors are both the source and destination nodes for the OPS), the resulting multiprocessor computer has very low bandwidth—only one processor may send a message in a slot. To alleviate this bandwidth problem a $POPS(d, g)$ network partitions the n processors into g groups of size d (d is also the degree of each coupler) each (so $n = dg$), and g^2 OPS couplers are used to interconnect pairs of processor groups. Specifically the groups are numbered 0 through $g-1$ and the source nodes for coupler $c(i,j)$ are the processors in group j; the destination nodes for this coupler are the processors in group i, $0 \leq i < g$, $0 \leq j < g$. Figure 2 shows how a $POPS(4,2)$ network is used to connect 8 processors. Destination processor i is the same processor as source processor i, $0 \leq i < 8$. A $POPS(4,2)$ network comprises $2^2 = 4$ degree $d = 4$ OPS couplers.

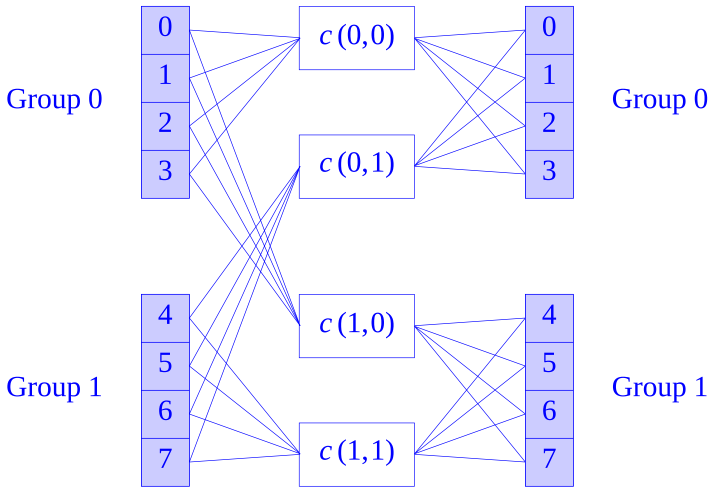

Figure 2: An 8-processor computer connected via a $POPS(4,2)$ network

When 8 processors are connected using a $POPS(8,1)$ network only one degree 8 OPS coupler is used. A 32 processor computer may be built using any one of the following networks: $POPS(32,1)$, $POPS(16,2)$, $POPS(8,4)$, $POPS(4,8)$, $POPS(2,16)$, and $POPS(1,32)$. A 25 processor computer may be built using a $POPS(25,1)$, $POPS(5,5)$, or $POPS(1,25)$ network. A multiprocessor computer that employs the POPS interconnection network is called a *POPS computer*.

The choice of the POPS network that is used to interconnect the processors affects both the interconnection cost as well as the bandwidth. When a $POPS(d,g)$ network is used to connect $n = dg$ processors, each processor must have g optical transmitters (one to transmit to each of the g OPSs for which it is a source node) and g receivers. The total number of transmitters and receivers is $2ng = 2n^2/d$, the number of OPSs is g^2, and each OPS has degree d. In one slot each OPS can receive a message from any one of its source nodes and broadcast this message to all of its destination nodes. In particular, in a single slot, a processor can send the same message to all of the OPSs for which it is a source node. In a single slot, a processor can receive a message from only one of the OPSs for which it is a destination node. (Melhem et al. [8] note that allowing a processor to receive different messages from different OPSs in the same slot permits faster all-to-all broadcast.)

A major advantage of the POPS network is that its diameter is 1. A message can be sent from processor i to processor j, $i \neq j$, in a single slot. Let $group(i)$ be the group that processor i is in. To send a meesage to processor j, processor i first sends the message to coupler $c(group(j), i)$. This coupler broadcasts the received message to all its destination processors; that is, to all processors in $group(j)$ [6, 8]. A one-to-all broadcast can also be done in one slot [6, 8]. Suppose that processor i wishes to broadcast a message to all other processors in the system. Processor i first sends the message to all couplers $c(*, group(i))$ for which it is a source node. Next all couplers of the form $c(*, group(i))$ broadcast the received message to their destination nodes. Since processor j is a destination node of coupler $c(group(j), group(i))$, processor j, $0 \leq j < n$ receives the message broadcast by processor i. This algorithm is easily extended to perform an all-to-all broadcast in n slots (or in 1 slot when a processor can receive, in a single slot, messages from all couplers that it is a destination node of). [6, 8] also give an algorithm for all-to-all personalized communication.

[6] shows how to embed rings and tori into POPS networks. [1] shows that POPS networks may be modeled by directed stack-complete graphs with loops. This modeling is used to obtain optimal embeddings of rings and de Bruijn graphs into POPS networks. An alternative multiprocessor interconnection network using multiple OPSs is proposed in [3].

In this paper we begin, in Section 2, by giving a feel for what is involved in programming a $POPS(d, g)$ computer for an application. The application we consider is matrix multiplication. In Section 3 we develop optimal routing algorithms for some of the commonly performed data permutations (e.g., intra group permutations, transpose, and vector reversal). Section 4 develops an efficient algorithm for the class of BPC permutations.

2 Matrix Multiplication

To get a feel for how we might program a POPS computer for an application, we consider the matrix multiplication problem. We make the following assumptions:

1. Two $N \times N$ matrices A and B are to be multiplied using a $POPS(d, g)$ computer with $n = dg = N^2$ processors.

2. A and B are mapped on to the computer one element per processor in row-major order. That is $A[i, j]$ and $B[i, j]$ are initially in processor $iN + j$, $0 \leq i < N$, $0 \leq j < N$. The result matrix C is to be similarly mapped on to the $n = N^2$ processors.

3. When $d \leq N$, d divides N, and when $d > N$, N divides d. This assumption ensures that a row of the matrix uses an integral number of processor groups or that a group contains an integral number of rows.

We use the notation $group(i, j)$ to refer to the processor group that contains elements $A[i, j]$ and $B[i, j]$ and that will eventually contain $c[i, j)$. $processor[i, j]$ refers to the processor that contains these three matrix elements. We also introduce a second numbering scheme for the processors of a $POPS(d, g)$ computer. Recall that in the first numbering scheme the processors are numbered 0 through $n - 1$ with processor i being the $i \bmod d$ processor in group $\lfloor i/d \rfloor$. The second numbering scheme is a two-dimensional scheme. $p(i, j)$ refers to processor j of group i. So processors $p(i, j)$ and $p(id + j)$ are the same.

The matrix multiplication is done using the algorithm of Figure 3. The algorithm is readily seen to be correct. It is also simpler than the algorithm used in [4] to multiply $N \times N$ matrices on mesh and hypercube computers that have $n = N^2$ processors.

```
$c[i,j) = 0,\ \ 0 \leq i < N,\ \ 0 \leq j < N$;
for (k = 0; k < N; k++)
{
    broadcast $A[i,k]$ from $processor[i,k]$ to $processor[i,*]$, $0 \leq i < N$;
    broadcast $B[k,j]$ from $processor[k,j]$ to $processor[*,j]$, $0 \leq j < N$;
    $c[i,j)$ += $A[i,k] * B[k,j],\ \ 0 \leq i < N,\ \ 0 \leq j < N$;
}
```

Figure 3: Parallel matrix multiplication algorithm

The CPU time is $N + 1$ units because it takes 1 unit to initialize all $c[i, j)$ values to 0 and another 1 unit in each iteration of the `for` loop to increment $c[i, j)$ by $A[i, k] * B[k, j]$ (note that the N^2 elements of C are initialized in parallel and also incremented in parallel). The CPU time may be reduced to N units by initializing $c[i, j) = A[i, 0] * B[0, j]$. This requires a slight restructuring of the algorithm and does not affect the time spent broadcasting the As and Bs.

The specifics of how the A and B values are broadcast and the complexity analysis for the broadcasts depends on whether $d = \sqrt{n} = N$, $d < N$, or $d > N$.

2.1 $d = \sqrt{n} = N$

The simplest case to consider is when each group holds exactly one row of each of the matrices A, B, and C. In this case $processor[i, j] = p(i, j)$.

The $A[i,k]$s, $0 \leq i < N$ are broadcast in one slot. First $processor[i,k]$ sends $A[i,k]$ to coupler $c(i,i)$. Next coupler $c(i,i)$ broadcasts $A[i,k]$ to $p(i,q) = processor[i,q]$, $0 \leq q < N$. $processor[i,q]$, $q \neq k$ receives the broadcast $A[i,k]$.

The broadcast of the $B[k,j]$s, $0 \leq j < N$ is done in two slots as follows. In the first slot $processor[k,j] = p(k,j)$ sends $B[k,j]$ to coupler $c(j,k)$, $j \neq k$. Next coupler $c(j,k)$ broadcasts the received $B[k,j]$ to $processor[j,*]$. The broadcast $B[k,j]$ is received by only one of the destination nodes of coupler $c(k,j)$; that is, by $p(j,k)$. In slot 2 $p(j,k)$, $0 \leq j < N$ sends the $B[k,j]$ value it received in slot 1 (if $j \neq k$) or its initial $B[k,j]$ value (in case $j = k$) to coupler $c(*,j)$ (i.e., to all couplers that it is a source node of). Next couplers $c(*,j)$ broadcast the received $B[k,j]$ value to their destination nodes and $p(*,j)$ receives $B[k,j]$.

An examination of how the $A[i,k]$ and $B[k,j]$ values are broadcast reveals that slot 1 of the $B[k,j]$ broadcast can be done at the same time as the $A[i,k]$ broadcast. This is because the $A[i,k]$ broadcast uses $p(*,k)$ as source nodes, couplers $c(i,i)$ for all i, and all processors other than $p(*,k)$ as destination nodes. Slot 1 of the $B[k,j]$ broadcast does not use any of these source nodes, couplers, or destination nodes. So there is no conflict in performing the slot 1 data routes of the $A[i,k]$ and $B[k,j]$ broadcasts in the same physical slot. Therefore each iteration of the matrix multiplication algorithm of Figure 3 takes only two slots. The total data routing time is $2N$ slots.

2.2 $d < N$

Since d divides N, no group has data from two or more rows of A, B, or C. The $A[i,k]$ broadcast is done in one slot by having $processor[i,k]$ send $A[i,k]$ to all couplers $c(q, group(i,k))$ such that group q contains elements of row i (i.e., $\lfloor q/(N/d) \rfloor = i$). Next couplers $c(q, group(i,k))$ broadcast the received $A[i,k]$ values to their destination nodes. $processor[i,j]$, $j \neq k$, receives the broadcast $A[i,k]$ (note that $processor[i,k]$ already has $A[i,k]$ and so need not be active in the receive part of the slot).

The $B[k,j]$ broadcast takes two slots. In the first slot $processor[k,j]$, $j \neq k$, sends $B[k,j]$ to coupler $c(group(j,k), group(k,j))$, which broadcasts $B[k,j]$ to all its destination nodes. Only the destination node $processor[j,k]$ receives the broadcast $B[k,j]$. Notice that $group(a,k) \neq group(b,k)$ for $a \neq b$. Therefore no coupler receives data from than one of its source nodes. In the second slot $processor[j,k]$ sends the received $B[k,j]$ (its initial $B[k,k]$ for $j = k$) to all couplers $c(*, group(j,k))$ for which it is a source node. Again, since $group(a,k) \neq group(b,k)$ for $a \neq b$, no coupler receives data from two or more source nodes. The couplers broadcast the $B[k,j]$ values received and $processor[i,j]$ receives the value broadcast by coupler $c(group(i,j), group(j,k))$.

It is easy to see that slot 1 of the B broadcast uses different source nodes and destination nodes than those used during the A broadcast. To see that the slot 1 couplers are different note that for $i \neq k$, $group(i,k) \neq group(k,j)$ and so the couplers used in slot 1 of the B broadcast are different from those used in the A broadcast. For $i = k$, the B broadcast uses couplers $c(group(j,k), group(k,j))$ and the A broadcast uses couplers $c(q, group(k,k))$ for $\lfloor q/(N/d) \rfloor = k$. If $j \neq k$ $group(j,k)$ does not have elements from row k, and so $group(j,k)$ cannot be one of the qs. Therefore, the couplers are different. For $j = k$, $group(j,k)$ equals one of the qs. However, coupler $c(group(j,k), group(k,j))$ with $j = k$ is not used in slot 1 of the B broadcast.

Consequently, slot 1 of the B broadcast may be done at the same time as slot 1 of the A broadcast. So the matrix multiplication is accomplished in $2N$ slots. Notice that the broadcast strategy used for the case $d = N$ is a special case of the strategy used when $d < N$.

2.3 $d > N$

When $d = N^2$ there is only one coupler. It takes one slot to broadcast each $A[i,k]$. So N slots are needed to broadcast all $A[i,k]$ values in each iteration of the `for` loop. Similarly N slots are needed to broadcast the $B[k,j]$ values. The total number of slots to complete the matrix multiply is $2N^2$.

Assume $N < d < N^2$. Now $g > 1$. Since N divides d, two or more rows share a group of processors. Since the processors of group i are the source nodes for $g - 1$ couplers (i.e., couplers $c(q,i)$, $q \neq i$) that are in groups other than i, at most $g - 1$ B values can be moved out of a group in one slot. In each iteration of the `for` loop all N values of row k must be transmitted to each of the remaining groups. This transmission cannot be done using fewer than $N/(g-1)$ slots. Therefore when the B broadcasts of each iteration are done independently from those of other iterations, we must use at least $N^2/(g-1)$ slots. Our algorithm uses $2N^2/g$ slots.

The number of rows in a group is $r = N/g$. In each iteration of the `for` loop r $A[i,k]$ values are to be broadcast within each group. This broadcast is done in r slots, one element per slot. In slot q, $processor[rs+q,k]$, sends $A[rs+q,k]$ to coupler $c(group(rs+q,k), group(rs+q,k))$, $0 \leq s < g$. The couplers broadcast the received As to all processors within their group, and processors $processor[rs+q,*]$ receive the $A[rs+q,k]$ value.

The $B[k,j]$ values are broadcast in batches of g elements using 2 slots per batch. In the first slot of batch q, $0 \leq q < r$, $processor[k, rs+q]$, $rs+q \neq k$, sends $B[k, rs+q]$ to coupler $c(group(rs+q,k), group(k, rs+q))$, $0 \leq s < g$. Coupler $c(group(rs+q,k), group(k,rs+q))$ sends the received B to $processor[rs+q,k]$. In the second slot for batch q $processor[rs+q,k]$ sends the B received in slot 1 (or its original B in case $rs + q = k$) to $c(*, group(rs + q, k))$. Then all couplers broadcast the received B values and $processor[i, rs + q]$ receives $B[k, rs + q]$ from $c(group(i, rs + q), group(rs + q, k))$, $0 \leq s < g$.

You may verify that slot 1 of the batch q broadcast of B and the slot q broadcast of A use different source nodes, couplers, and destination nodes. Therefore, these two operations may be done in the same slot. Consequently, a total of $2r = 2N/g$ slots per `for` loop iteration are needed to do the A and B broadcasts. Over all iterations of the `for` loop, $2N^2/g$ slots are used.

2.4 Comparing The Three Cases

We may divide the total time taken by our parallel matrix multiplication algorithm into two parts—(1) the time spent in the broadcast steps and (2) the time spend in the compute steps (i.e., the remainder of the algorithm). The time for part (2) is independent of the coupler degree d. When $d \leq N$, our algorithm spends $2N$ slots doing the data broadcasts. Since a smaller d requires more couplers (number of couplers $= g^2 = (n/d)^2 = N^4/d^2$) and more transmitters and receivers (number of transmitters = number of receivers $= n^2/d$) there is no advantage to having $d < \sqrt{n} = N$ (as far as our matrix multiplication algorithm is concerned).

When $d \geq N$ the number of slots needed for the A and B broadcasts increases as d increases. This increase in number of slots is, of course, accompanied by a decrease in hardware cost (as measured by the number of couplers, transmitters, and receivers). In fact when $d = n = N^2$ the communication cost (i.e., the broadcast cost) of our algorithm is $2N^2$ slots whereas the computation cost is only $O(N)$. Computation and communication costs are balanced when $d = \sqrt{n} = N$.

3 Frequently Performed Data Rearrangements

In this section we develop efficient algorithms to perform the following data permutations: permute the data in a single group, perform intragroup permutations within all groups, vector reversal, and matrix transpose.

Although [6, 8] give a data permutation algorithm for the $POPS(d, g)$ network, this algorithm is optimal only when we require that each data item be routed from its source processor to its destination processor using a single slot (or hop). Under this assumption datum initially in $p(i)$ and destined for $p(\sigma(i))$, where $\sigma(i)$ is the permutation that is to be performed and $i \neq \sigma(i)$, must be routed using the path $p(i) \rightarrow c(group(\sigma(i)), group(i)) \rightarrow p(\sigma(i))$. When an intragroup permutation with $i \neq \sigma(i)$ for all i in the group (an example of such a permutation would be a nonzero intragroup shift), for example, is performed we get a coupler conflict—all d processors of a group need the same coupler to get their source data to the destination processors. This coupler conflict is eliminated by scheduling the use of the couplers so that no coupler has more than one source node transmitting data to the coupler. For an intragroup pemutation with $i \neq \sigma(i)$ for all i d time slots are taken to complete the permutation with each data making a single hop. We refer to the permutation algorithm of [6, 8] as the *single-hop* algorithm.

3.1 Single Intragroup Permutation

As noted above, when $i \neq \sigma(i)$ for all i in the group being permuted, the single hop algorithm takes d slots to perform an intragroup permutation. Without loss of generality, assume that the data in group 0 are to be permuted as per the permutation σ_0. That is, data from $p(0, i)$ is to be moved to $p(0, \sigma_0(i))$, $0 \leq i < d$. The single intragroup permutation σ_0 may be performed in $2\lceil d/g \rceil$ slots by routing g elements to their destination processors in every pair of slots. The number of slot pairs is d/g and in the qth pair, $0 \leq q < d/g$, processors $qg + i$, $0 \leq i < g$ send their data to the destination processors using two slots as follows:

1. In the first slot of a slot pair $p(0, qg + i)$ sends its data to $p(i, 0)$ using the path $p(0, qg + i) \rightarrow c(i, 0) \rightarrow p(i, 0)$, $0 \leq i < g$.

2. In the second slot of the slot pair $p(i, 0)$ sends the data it received in the first slot using the path $p(i, 0) \rightarrow c(0, i) \rightarrow p(0, \sigma_0(qg + i))$, $0 \leq i < g$.

The correctness of the above algorithm is easily established. As was the case for the matrix multiplication algorithm of Section 2, there is no advantage to having $d < \sqrt{n}$ because when $d < \sqrt{n}$, $g > \sqrt{n}$ and the intragroup permutation takes 2 slots; the same as when $d = g = \sqrt{n}$.

Although the above intragroup algorithm is simple, it is not optimal (except when $d \leq \sqrt{n}$). Figure 4 gives an optimal algorithm.

Once again, correctness is easily established. The number of slots is seen to be $\lceil (d-1)/g \rceil + 1$. Notice that each data item is routed to its destination using either one or two hops. Under the assumption that $i \neq \sigma_0(i)$ for any i in group 0, the single hop algorithm takes d slots. So the algorithm of Figure 4 provides a speedup of $d/(\lceil (d-1)/g \rceil + 1)$. When $d = g = \sqrt{n} > 1$ the speedup is $d/2$. When $g = 1$ the single hop algorithm uses the same number of slots (i.e., d) as used by the algorithm of Figure 4.

A circular shift of the elements of group 0 by one unit takes d slots when the single hop algorithm is used and $\lceil (d-1)/g \rceil + 1$ slots when we use the algorithm of Figure 4.

1. In slot 0 $p(0,0)$ uses the path $p(0,0) \rightarrow c(0,0) \rightarrow p(0,\sigma_0(0))$ to get its data to its destination processor. Processors $p(0,i)$, $0 < i < g$ use the paths $p(0,i) \rightarrow c(i,0) \rightarrow p(i,0)$ to get their data to intermediate processors in groups other than group 0.
2. In slot q, $q > 0$ we do the following:
   (a) $p(0,qg)$ uses the path $p(0,qg) \rightarrow c(0,0) \rightarrow p(0,\sigma_0(qg))$ to get its data to its destination processor.
   (b) Processors $p(i,0)$, $i \neq 0$, use the following paths to get the data they received in slot $q-1$ to destination processors: $p(i,0) \rightarrow c(0,i) \rightarrow p(0,\sigma_0((q-1)g+i))$.
   (c) Processors $p(0,qg+i)$, $1 \leq i < g$ and $qg+i < d$, use the following paths to get their data to intermediate processors: $p(0,qg+i) \rightarrow c(i,0) \rightarrow p(i,0)$.

Figure 4: An optimal algorithm for a single group intragroup permutation

Theorem 1 *The algorithm of Figure 4 is optimal when $i \neq \sigma_0(i)$, $0 \leq i < d$.*

Proof: Since all the data that is to be routed begins and ends in group 0, at most 1 datum can be routed to its final destination in the first slot of any algorithm. Since group 0 has g couplers, at most g different data can be routed to the group 0 processors in any slot. So every algorithm routes at most 1 datum to the final destination in the first slot and at most g in each of the remaining slots. Since d data are to be routed to their destination processors, at least $\lceil (d-1)/g \rceil + 1$ slots must be used. ■

3.2 Multiple Intragroup Permutations

Suppose that data from $p(i,j)$ is to be routed to $p(i,\sigma_i(j))$, $0 \leq i < g$, $0 \leq j < d$. The $2\lceil d/g \rceil$-slot algorithm of Section 3.1 may be run, in parallel, on all groups. Therefore, data in all groups may be permuted in $2\lceil d/g \rceil = 2\lceil n/g^2 \rceil$ slots. We can reduce the number of slots to $2\lceil n/(g+g^2) \rceil$ by moving batches of $g+g^2$ elements to their destination processors using 2 slots per batch. In the first slot g elements are moved to their destination processors and g^2-g elements are moved to intermediate processors. In the second slot g^2 elements are moved to their destination processors. Figure 5 gives the algorithm.

Using the algorithm of Figure 5 we can, for example, perform a circular shift by 1 of data in each group using $2\lceil n/(g+g^2) \rceil$ slots. Shifting all groups by 1 takes $d = n/g$ slots when the single hop algorithm is used.

Theorem 2 *The algorithm of Figure 5 is within 1 slot of being optimal when $j \neq \sigma_i(j)$, $0 \leq i < g$, $0 \leq j < d$.*

```
1. Do the following for $0 \leq q < \lceil n/(g+g^2) \rceil$

2. // slot 1 of slot pair $q$
   Let $h = g + 1$.
   $p(i, qh)$ uses the path $p(i, qh) \rightarrow c(i, i) \rightarrow p(i, \sigma_i(qh))$ to route its data to the destination processor.
   Processors $p(i, qh + s)$, $1 \leq s < g$ use the following paths to route their data to intermediate
   processors (let $t = (i + s) \bmod g$): $p(i, qh + s) \rightarrow c(t, i) \rightarrow p(t, i)$ when $\sigma_t(qh) \neq i$ and $p(i, qh + s) \rightarrow$
   $c(t, i) \rightarrow p(t, t)$ when $\sigma_t(qh) = i$.

3. // slot 2 of slot pair $q$
   $p(i, qh + g)$ uses the path $p(i, qh + g) \rightarrow c(i, i) \rightarrow p(i, \sigma_i(qh + g))$ to route its data to the destination
   processors.
   Processors that received data from a different group in slot 1 of this slot pair route the received
   data via the coupler in the destination group to the destination processor.
```

Figure 5: Optimal algorithm to permute in all groups

Proof: Suppose that an algorithm completes the group permutations in k slots. A total of n elements are to be moved from their initial processors to their destination processors. At most kg of these elements can be so moved using a single hop. Therefore, at least $n - kg$ elements are moved using 2 or more hops each. So the total number of data moves (in one data move a data item makes 1 hop) is $\geq 2(n - kg) + kg = 2n - kg$. In a $POPS(d, g)$ at most g^2 data moves can be made in one slot. Therefore, $kg^2 \geq 2n - kg$. So $k \geq 2n/(g + g^2)$. Since k is an integer, $k \geq \lceil 2n/(g + g^2) \rceil$.

The number of slots used by the algorithm of Figure 5 is $2\lceil n/(g + g^2) \rceil \leq \lceil 2n/(g + g^2) \rceil + 1$. ■

3.3 Vector Reversal

Data from $p(i)$ is to be moved to $p(n-1-i)$, $0 \leq i < n$. When a $POPS(d, g)$ is used we must move data from $p(i, j)$ to $p(g - 1 - i, d - 1 - j)$, $0 \leq i < g$, $0 \leq j < d$. The single-hop algorithm requires d slots to do a vector reversal. A multi-hop algorithm can perform a vector reversal in $2\lceil d/g \rceil$ slots. Furthermore, in the multi-hop algorithm no data makes more than 2 hops.

For the multi-hop algorithm it is convenient to consider the three cases ($d = g = \sqrt{n}$, $d < \sqrt{n}$, and $d > \sqrt{n}$) that we considered for the matrix multiplication problem of Section 2. First consider the case $d = g = \sqrt{n}$. When $d = g = 1$ no work is to be done as $n = 1$. So assume that $d > 1$. The reversal can be accomplished in $2d/g = 2$ slots using the routing

$$p(i, j) \rightarrow c(j, i) \rightarrow p(j, i) \rightarrow c(g - 1 - i, j) \rightarrow p(g - 1 - i, d - 1 - j)$$

When $d > 1$ at least 2 slots are needed for the reversal because in the first slot only one processor of group 0 can route its data to the destination group. Therefore the reversal method just described is optimal.

Next consider the case $d < \sqrt{n}$. Now $g > \sqrt{n}$. If $d \leq 2$ the single-hop algorithm may be used to reverse the elements in a single slot. Notice that when $d = 1$ every reversal algorithm must use at least one hop, and when $d = 2$ group 0 can send only one datum to the proper group in the first slot and so every reversal algorithm must use at least 2 slots. When $d > 2$ the following 2 slot routing may be used to perform the reversal:

$$\begin{aligned}
p(i,j) &\rightarrow c((id+j) \bmod g, i) && (1)\\
&\rightarrow p((id+j) \bmod g, \lfloor (id+j)/g \rfloor) && (2)\\
&\rightarrow c(g-1-i, (id+j) \bmod g) && (3)\\
&\rightarrow p(g-1-i, d-1-j) && (4)
\end{aligned}$$

To establish the correctness of the above 2 slot routing algorithm we note that data that start in two different processors $p(i_1, j_1)$ and $p(i_2, j_2)$ use different couplers in slot 1 (i.e., for the source node to coupler routing of line (1)). For this observe that when $i_1 \neq i_2$ the slot 1 couplers differ in their second index. When $i_1 = i_2$, $j_1 \neq j_2$. Now since j_1 and j_2 are between 0 and $d-1$ and since $d < g$, $(i_1 d + j_1) \bmod g \neq (i_2 d + j_2) \bmod g$.

Next note that the slot 1 destination nodes (see line (2)) for data originating in different processors is different. For this observe that $i_1 d + j_1 \neq i_2 d + j_2$. So if $(i_1 d + j_1) \bmod g = (i_2 d + j_2) \bmod g$ then $\lfloor (i_1 d + j_1)/g \rfloor \neq \lfloor (i_2 d + j_2)/g \rfloor$. Finally for line (3) observe that when $i_1 \neq i_2$ the slot 2 couplers differ in their first index. When $i_1 = i_2$, $j_1 \neq j_2$. Now since j_1 and j_2 are between 0 and $d-1$ and since $d < g$, $(i_1 d + j_1) \bmod g \neq (i_2 d + j_2) \bmod g$.

Also note that when $d > 2$ every reversal algorithm must use at least 2 slots. Therefore, our algorithm for the case $d < \sqrt{n}$ is also optimal. If we put $d = g = \sqrt{n}$ into the algorithm for the case $d < \sqrt{n}$, we get the algorithm given earlier for $d = \sqrt{n}$.

The final case has $d > \sqrt{n}$. When $d \geq n/2$, $g \leq 2$ and only one datum can be routed to its destination processor in each slot. So the single-hop algorithm is optimal. For $d < n/2$ we can do the reversal in $2\lceil d/g \rceil$ slots, which is an improvement over the single-hop algorithm. Our $2\lceil d/g \rceil$ slot algorithm (Figure 6) does the reversal in $\lceil d/g \rceil$ iterations of 2 slots each. In each iteration g elements are correctly routed from their source processors to their destination processors.

```
1. Do the following for $0 \le q < \lceil d/g \rceil$
2. // slot 1 of slot pair $q$
   $p(i, qg)$ uses the path $p(i, qg) \rightarrow c(g - i - 1, i) \rightarrow p(g - 1 - i, d - 1 - qg)$ to route its data to the destination processor.
   Processors $p(i, qg + s)$, $1 \le s < g$, $(i + s) \bmod g \ne g - i - 1$ route their data to intermediate processors using the paths $p(i, qg + s) \rightarrow c((i + s) \bmod g, i) \rightarrow p((i + s) \bmod g, s)$.
3. // slot 2 of slot pair $q$
   $p(i, qg+s)$, $(i+s) \bmod g = g-1-i$ uses the path $p(i, qg+s) \rightarrow c(g-i-1, i) \rightarrow p(g-i-1, d-1-qg-s)$ to route its data to the destination processor.
   Processors $p((i+s) \bmod g, s)$, $1 \le s < g$, $(i+s) \bmod g \ne g-i-1$ route the data they received in slot 1 to their destination processors using the paths $p((i+s) \bmod g, s) \rightarrow c(g-i-1, (i+s) \bmod g) \rightarrow p(g-i-1, d-1-qg-s)$.
```

Figure 6: Algorithm to reverse when $\sqrt{n} < d < n/2$

The correctness and complexity of Figure 6 are easily verified. The following theorem establishes the near optimality of Figure 6.

Theorem 3 *Suppose that $\sqrt{n} < d < n/2$. When g is even, at least $\lceil 2d/g \rceil$ slots are needed to do a reversal, and when g is odd, at least $\lceil 2n/(g^2+1) \rceil = \lceil 2d/(g+1/g) \rceil$ slots are needed.*

Proof: In any k-slot reversal algorithm at most kg elements are routed to their destination processors using one hop per element. Therefore, at least $n - kg$ elements are routed using 2 or more hops per element. Therefore, the number of data moves is at least $2(n - kg) + kg = 2n - kg$. When g is even we may assume that there are no intragroup hops; intragroup hops are of no benefit when g is even because no element has the same initial and final group. So the number of data moves made in a slot is at most $g^2 - g$. Therefore, in k slots at most $(g^2 - g)k$ data moves are made. For the algorithm to be correct, $(g^2 - g)k \ge 2n - kg$. So $k \ge 2n/g^2 = 2d/g$. Since k is an integer, we get $k \ge \lceil 2d/g \rceil$.

When g is odd intragroup hops benefit only the middle group $\lceil g/2 \rceil$ because the source and destination groups for elements in this group are the same. Therefore, we may assume that intragroup hops occur only in group $\lceil g/2 \rceil$. So in each slot at most $g^2 - g + 1$ data moves are made. Therefore, for a correct reversal algorithm $(g^2 - g + 1)k \ge 2n - kg$. So $k \ge 2n/(g^2+1) = 2d/(g + 1/g)$. Since k is an integer, we get $k \ge \lceil 2d/(g+1/g) \rceil$. ■

3.4 Matrix Transpose

As in Section 2 assume that an $N \times N$ matrix, $N > 1$, is mapped into an n processor $POPS(d, g)$ in row-major order and that either d divides N or g divides N. Under these assumptions the single-hop permutation algorithm of [6, 8] performs a matrix transpose in $\lceil d/g \rceil$ slots. This is the minimum number of slots in which a $POPS(d, g)$ can transpose a matrix. Therefore, the single-hop algorithm of [6, 8] is optimal for matrix transpose.

Theorem 4 *Under the stated assumptions a $POPS(d, g)$ must use at least $\lceil d/g \rceil$ slots to do a matrix transpose.*

Proof: When $1 \le d \le N = \sqrt{n}$, $d \le g$. So $\lceil d/g \rceil = 1$. Since $N > 1$ at least 1 matrix element is to move into a different processor. Therefore every transpose algorithm must use at least 1 slot.

When $d > N$ each group contains $r = N/g$ rows of the matrix. $Nr - r^2$ elements must leave each group as a result of the transpose. In a slot at most $g - 1$ elements may leave a group. Therefore, every transpose algorithm must use at least $(Nr - r^2)/(g - 1) = (d - r^2)/(g - 1) = (d - d/g)/(g - 1) = d/g$ slots. ■

4 BPC Permutations

Lenfant [7] identified a few permutations (e.g., transpose, bit reversal, vector reversal) as permutations that arise frequently in applications. When n is a power of 2 these permutations can be described by a rearrangement of the bits in the source processor index. That is $\sigma(i)$ is obtained by permuting the bits of i in the same way for all i. For example, in the bit reversal permutation data in $p(i)$ is to be routed to $p(\sigma(i))$, where $\sigma(i)$ is obtained by reversing the bits in the binary representation of i. When $n = 16$, the binary representation of i is $i_3 i_2 i_1 i_0$, and $\sigma(i) = i_0 i_1 i_2 i_3$.

Nassimi and Sahni [9] defined the class of bit-permute-complement permutations (BPC permutations) in which each σ is given as a permutation of the bits of the source indexes, the permutation of the bits may also be accompanied by the complementing of some or all bits. The BPC class of permutations includes all of the frequently occurring permutations given in [7].

A BPC permutation [9] is specified by a vector $A = [A_{p-1}, A_{p-2}, \ldots, A_0]$ where

(a) $A_i \in \{\pm 0, \pm 1, \ldots, \pm (p-1)\}$, $0 \le i < p$ and

(b) $[|A_{p-1}|, |A_{p-2}|, \ldots, |A_0|]$ is a permutation of $[0, 1, \ldots, p-1]$.

Source		Destination	
Processor	Binary	Binary	Processor
0	0000	1001	9
1	0001	0001	1
2	0010	1101	13
3	0011	0101	5
4	0100	1011	11
5	0101	0011	3
6	0110	1111	15
7	0111	0111	7
8	1000	1000	8
9	1001	0000	0
10	1010	1100	12
11	1011	0100	4
12	1100	1010	10
13	1101	0010	2
14	1110	1110	14
15	1111	0110	6

Figure 7: Source and destination of the BPC permutation $[-0, 1, 2, -3]$

The destination for the data in any processor may be computed in the following way. Let $s_{p-1}s_{p-2}\dots s_0$ be the binary representation of the processor's index. Let $d_{p-1}d_{p-2}\dots d_0$ be that of the destination processor's index. Then,

$$d_{|A_i|} = \begin{cases} s_i & if \quad A_i \geq 0, \\ 1 - s_i & if \quad A_i < 0. \end{cases}$$

In this definition, -0 is to be regarded as < 0, while $+0$ is ≥ 0.

When $n = 16$ the processor indexes have four bits. The BPC permutation $[-0, 1, 2, -3]$ requires data from each processor $s_3s_2s_1s_0$ be routed to processor $(1 - s_0)s_1s_2(1 - s_3)$. Figure 7 lists the source and destination processors of the permutation.

Figure 8 gives the permutation vector A for several frequently occurring permutations.

When dealing with BPC permutations we require that n be a power of 2. This requirement implies that d and g are also powers of 2.

For the bit-reversal permutation the single-hop algorithm of [6, 8] takes $\lceil d/g \rceil$ slots to complete a bit reversal. This number of slots may be shown to be optimal (even for multi-hop algorithms). The proof of this result is the same as that of Theorem 4.

Before developing a multi-hop routing algorithm for the class of BPC permutations, we develop algorithms for the perfect shuffle and bit shuffle permutations. These algorithms are developed only for

Permutation	Permutation Vector
Transpose	$[p/2-1,\ldots,0,p-1,\ldots,p/2]$*
Perfect Shuffle	$[0,p-1,p-2,\ldots,1]$
Unshuffle	$[p-2,p-3,\ldots,0,p-1]$
Bit Reversal	$[0,1,\ldots,p-1]$
Vector Reversal	$[-(p-1),-(p-2),\ldots,-0]$
Bit Shuffle	$[p-1,p-3,\ldots,1,p-2,p-4,\ldots,0]$*
Shuffled Row-major	$[p-1,p/2-1,p-2,p/2-2,\ldots,p/2,0]$*

* p is even

Figure 8: Permutations and their permutation vectors

the case $d = g = \sqrt{n}$. The ideas used in these algorithms are then used to obtain a routing algorithm for all BPC permutations for all d and g.

4.1 Perfect Shuffle, $d = g = \sqrt{n}$

When $d = g = 1$ no work is to be done, and when $d = g = 2$ a perfect shuffle is the same as a transpose. So assume that $d = g > 2$ and a power of 2. The perfect shuffle permutation is realized in 2 slots using the following routing:

$$
\begin{aligned}
p(i,j) &\rightarrow c((i+j) \bmod g, i) && (5)\\
&\rightarrow p((i+j) \bmod g, j) && (6)\\
&\rightarrow c(i'', (i+j) \bmod g) && (7)\\
&\rightarrow p(i'', j'') && (8)
\end{aligned}
$$

where $p(i'', j'')$ is the destination for data initially in $p(i,j)$.

Note that two processors cannot send their data to the same coupler in line 5 because if two processors have different i values the couplers used differ in their second index. If two processors have the same i value (i.e., they are in the same group), then since $0 \leq j < d = g$, the couplers must differ in their first index. To see that the destination processors in line (6) are different note that data that start in processors that have different second index use destination processors that differ in their second index; data that start in processors with the same second index start in processors with a different first index and this first index is in the range 0 through $g-1$. Therefore, the destination processors of line (6) must differ in their first index.

Let $a = \log_2 g - 1$, and let c_d denote the dth (least significant) bit of c. For the perfect shuffle permutation, $i'' = \lfloor i/2 \rfloor + j_0 2^a$. Data elements initially in different processors $p(u,v)$ and $p(x,y)$ cannot reach the same coupler in line (7). To see this observe that if $u = x$, the data are in different groups in line (6) and so the couplers used in line (7) differ in their second index. Suppose $u \neq x$ and $i''_1 = i''_2$. Now since $i''_1 = \lfloor u/2 \rfloor + (v)_0 2^a$ and $i''_2 = \lfloor x/2 \rfloor + (y)_0 2^a$, $(v)_0 = (y)_0$ and $\lfloor u/2 \rfloor = \lfloor x/2 \rfloor$. Therefore, $u = x \pm 1$. So $u+v$ and $x+y$ differ in bit 0. From this and the fact that g is a power of 2 it follows that $(u+v) \bmod g \neq (x+y) \bmod g$.

The described perfect shuffle algorithm is optimal when $d = g > 2$ and a power of 2. Since the minimum d is 4, at least 2 elements are to be moved from group 0 to group $d/2$. Only 1 element can be moved from group 0 to group $d/2$ in the first slot. Therefore, at least 2 slots are needed to do a perfect shuffle when $d = g > 2$ and a power of 2.

4.2 Bit Shuffle, $d = g = \sqrt{n}$

When $d = g = 1$ the bit shuffle is not defined, and when $d = g = 2$ no data are to be moved. So assume that $d = g > 2$ and a power of 2. The bit shuffle permutation is realized in 2 slots using a routing similar to that used for a perfect shuffle. This routing is given below:

$$
\begin{aligned}
p(i,j) &\rightarrow c(i',i) && (9)\\
&\rightarrow p(i',j) && (10)\\
&\rightarrow c(i'',i') && (11)\\
&\rightarrow p(i'',j'') && (12)
\end{aligned}
$$

where $p(i'',j'')$ is the destination for data initially in $p(i,j)$, and $p(i',j)$ is the intermediate processor used for data initially in $p(i,j)$. For the perfect shuffle algorithm of Section 4.1, $i' = (i+j) \bmod g$.

To determine i' for the bit shuffle permutation, do the following:

1. Let $a = ig + j$.

2. Number the bits in a from right to left beginning with the number 0. The bits that have an odd number are the *odd bits* of a. The remaining bits are the *even bits* of a.

3. Let b be the number whose bits are the odd bits in the group index i followed by the even bits in i. For definiteness, we preserve the relative order of the odd bits and the even bits.

4. Let c be the number whose bits are the even bits in j followed by the odd bits in j. For definiteness, we preserve the relative order of the odd bits and the even bits.

5. $i' = (b + c) \bmod g$.

When $n = 64$ a processor index is given by 6 bits $i_5 i_4 i_3 i_2 i_1 i_0$. Bits i_5, i_3, and i_1 are the odd bits, and bits i_4, i_2, and i_0 are the even bits. When $d = g = \sqrt{n} = 8$, $p(i, j)$ has $i = i_5 i_4 i_3$ and $j = i_2 i_1 i_0$. To compute i' we determine $b = (i_5 i_3 i_4)_2$ and $a = (i_2 i_0 i_1)_2$. Then we get $i' = (b+c) \bmod 8$. If $i = (001)_2 = 1$ and $j = (110)_2 = 6$, then $b = (010)_2 = 2$, $c = (101)_2 = 5$, and $i' = (2 + 5) \bmod 8 = 7$.

The proof that data that start in different processors use different couplers in line (9) and different intermediate processors in line 10 is the same as that for the perfect shuffle permutation. Therefore, we show only that data that start in different processors use different couplers in line (11). Data that start in different processors of the same group i get to intermediate processors that are in different groups in line (10). Therefore, these data use couplers in line (11) that differ in their second index. Consider data originating in $p(i_1, j_1)$ and $p(i_2, j_2)$, $i_1 \neq i_2$. If these data are destined for processors in different groups then the line (11) couplers differ in the first index. So assume that these two data are destined for processors in the same group. Let $odd(i_1)$ be the sequence of odd bits in i_1 and let $even(i_1)$ be the sequence of even bits in i_1. Let $A = odd(i_1)even(i_1) + even(j_1)odd(j_1)$ and $B = odd(i_2)even(i_2) + even(j_2)odd(j_2)$. The destination groups for data originating in $p(i_1, j_1)$ and $p(i_2, j_2)$ are $odd(i_1)odd(j_1)$ and $odd(i_2)odd(j_2)$, respectively. Since the destination groups are the same, $odd(i_1) = odd(i_2)$ and $odd(j_1) = odd(j_2)$. Now since $i_1 \neq i_2$, $even(i_1) \neq even(i_2)$. If $even(j_1) = even(j_2)$, then $0 < |A - B| < g$. Consequently, $i'_1 \neq i'_2$ and the line (11) couplers differ in their first index. If $even(j_1) \neq even(j_2)$, then $|even(j_1)odd(j1) - even(j_2)odd(j_2)|$ is a nonzero multiple of 2^q, where q is the number of bits in $odd(j_2)$ (which is the same as the number of bits in $even(i_2)$). This difference is also less than g. So this difference is at most $g - 2^q$ (note that 2^q divides g because g is a larger power of 2). Since $|odd(i_1)even(i_1) - odd(i_2)even(i_2)| = |odd(i_1)even(i_1) - odd(i_1)even(i_2)| < 2^q$, $0 < |A - B| < g - 2^q + 2^q = g$. Therefore, $i'_1 = A \bmod g \neq i'_2 = B \bmod g$.

When $d = g > 2$ and a power of 2, a bit shuffle requires at least 2 processors of group 0 to send data to group 1. In the first slot of any routing algorithm at most 1 group 0 processor can send data to a group 1 processor. So every bit shuffle algorithm must use at least 2 slots. Therefore, the bit shuffle algorithm of lines 9 through 12 is optimal.

4.3 General BPC Permutations

We may generalize the $d = g = \sqrt{n}$ routing algorithms developed in Sections 4.1 and 4.2 to obtain a 2 slot routing algorithm for every BPC permutation. The generalization has the structure of lines 9 through 12. Only the choice of i' varies with the specific BPC permutation that is to be performed.

To arrive at the proper i', examine the i' used in Sections 4.1 and 4.2 for the perfect shuffle and bit shuffle permutations. In both cases i' is given by an expression of the form $(a+b) \bmod g$. For the perfect shuffle permutation $a = i$ and $b = j$. For the bit shuffle permutation a and b are obtained by separating the bits in i and j into two parts—odd bits (these are the bits of i and j that define the value of i'') and even bits (these are the bits of i and j that define to the value of j''). Since the odd bits of i contribute to the value of i'' we refer to these as the *stay bits* (they stay over from the source group index into the destination group index). The even bits of i are called the *go bits.* The odd bits of j are the go bits of j and js even bits are its stay bits. For the bit shuffle permutation $i' = (stay(i)go(i) + stay(j)go(j)) \bmod g$, where $stay(i)$ is the sequence of stay bits in i and $go(i)$ is the sequence of go bits. For definiteness we require that the relative order of the stay bits and the go bits be unchanged (this is not required by the correctness proof provided for the bit shuffle algorithm).

The stay and go interpretation of the method used to compute i' for the bit shuffle permutation applies even to the perfect shuffle method. For the perfect shuffle permutation, for any processor $p(i, j)$, $go(i)$ is the least significant bit of i and $go(j)$ is the least significant of j. Therefore, $stay(i)go(i) = i$ and $stay(j)go(j) = j$.

Fortunately for us, using $i' = (stay(i)go(i) + stay(j)go(j)) \bmod g$ works for all BPC permutations! The proof that there is no coupler conflict in line (9) and no intermediate node conflict in line (10) is the same as for the perfect shuffle permutation, and the proof that there is no coupler conflict in line (11) is the same as for the bit shuffle permutation.

Even though the generalized BPC routing algorithm uses only 2 slots for each permutation, the algorithm is not optimal for every BPC permutation. For example, the single-hop scheme of [6, 8] does a transpose in 1 slot when $d = g = \sqrt{n}$ and the identity permutation can be done requires no routing. The generalized scheme is, however, optimal for every BPC permutation that requires two or more elements to be moved from a single source group to a single destination group.

An examination of the routing of lines 9—12 reveals that this routing works even when $d < \sqrt{n}$. When $d > \sqrt{n}$ a BPC permutation may be done in $2d/g$ slots by running the algorithm of lines 9-12 d/g times. We refer to each run of this algorithm as a phase. In each phase exactly g processors per group (for a total of g^2 processors) route their data to destination processors. The leftmost $r = \log_2(d/g)$ $stay(j)$ bits of these g^2 processors are the same. So when $d = 16$ and $g = 4$, the two leftmost bits of $stay(j)$ are the same for the 16 processors that route their data to destination processors in a given phase. Notice that $r = \log_2(d/g) = \log_2 d - \log_2 g = \#bits(stay(j)) + \#bits(go(j)) - \#bits(i) \leq \#bits(stay(j))$, where $\#bits()$ denotes the number of bits). The proof that no phase has a coupler or intermediate processor conflict is similar to that used to show the absence of conflicts in the routing algorithms for the perfect shuffle and bit shuffle permutations. Note that the algorithm for the case $d = g = \sqrt{n}$ is the same as the algorithm just described when r is set to $0 = \log_2(d/g)$.

5 Conclusion

We have developed optimal and near optimal algorithms for matrix multiplication, commonly occurring data permutations, and BPC permutations on a $POPS(d, g)$ computer. Our research shows that the POPS interconnection network is a powerful network for which optimal and near-optimal algorithms may be deveoped with modest effort.

References

[1] P. Berthomé and A. Ferreira. Improved embeddings in POPS networks through stack-graph models. *Third International Workshop on Massively Parallel Processing Using Optical Interconnections*, IEEE, 130-135, 1996.

[2] D. Chiarulli, S. Levitan, R. Melhem, J. Teza, and G. Gravenstreter. Multiprocessor interconnection networks using partitioned optical passive star (POPS) topologies and distributed control. *First International Workshop on Massively Parallel Processing Using Optical Interconnections*, IEEE, 70-80, 1994.

[3] D. Coudert, A. Ferreira, and X. Muñoz. Multiprocesor architectures using multi-hop multi-ops lightwave networks and distributed control *12th International Parallel processing Symposium and 9th Symposium on Parallel and Distributed Processing*, IEEE, 151-155, 1998.

[4] E. Dekel, D. Nassimi, and S. Sahni, Parallel matrix and graph algorithms. *SIAM Jr. on Comp.*, 10, 4, 657-675, 1981.

[5] G. Graventreter, R. Melhem, D. Chiarulli, S. Levitan, and J. Teza. The partitioned optical passive stars (POPS) topology. *9th International Parallel processing Symposium*, IEEE, 4-10, 1995.

[6] G. Graventreter and R. Melhem. Realizing common communication patterns in partitioned optical passive star (POPS) networks. *IEEE Transactions on Computers*, 998-1013, 1998.

[7] J. Lenfant. Parallel permutations of data: A Benes network control algorithm for frequently used permutations. *IEEE Transactions on Computers*, 27, 7, 637-647, 1978.

[8] R. Melhem, G. Graventreter, D. Chiarulli, and S. Levitan. The communication capabilities of partitioned optical passive star networks. In *Parallel computing using optical interconnections*, K. Li, Y. Pan, and S. Zheng, Editors, Kluwer Academic Publishers, 77-98, 1998.

[9] D. Nassimi and S. Sahni, An Optimal Routing Algorithm for Mesh-Connected Parallel Computers. *Jr. of the ACM*, 27, 1, 6-29, 1980.