The Partitioned Optical Passive Stars Network: Simulations And Fundamental Operations

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

Abstract

We show how a multiprocessor computer interconnected by a partitioned optical passive stars network (POPS) can simulate hypercube and mesh-connected computers. POPS algorithms for data sum, prefix sum, rank, adjacent sum, consecutive sum, concentrate, distribute, and generalize are also developed. These fundamental operations form the building blocks of parallel algorithms for many applications.

Keywords: Partitioned optical passive stars network, hypercube and mesh simulation, fundamental arithmetic and data movement operations.

1 Introduction

The partitioned optical passive stars network (POPS) was proposed in [3, 5, 6, 9] 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*.

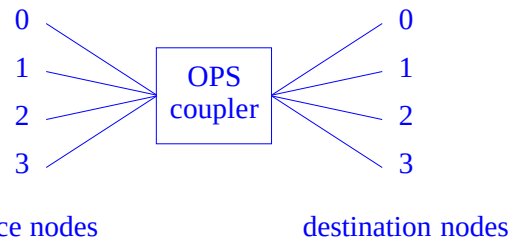

Figure 1: An optical passive star coupler with 4 source and 4 destination nodes

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.

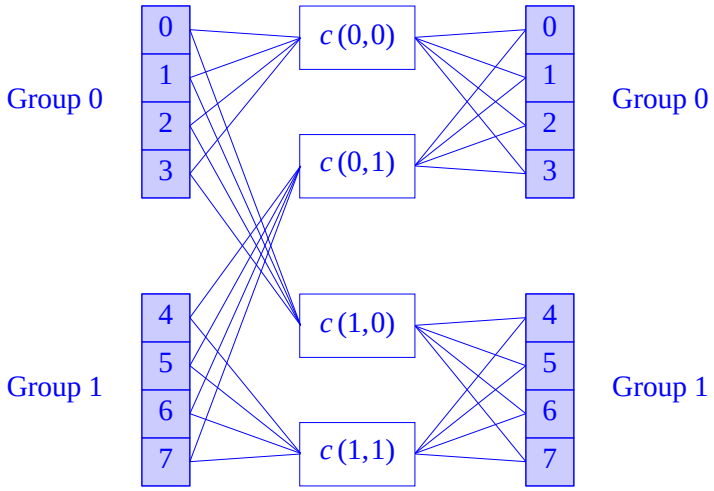

Figure 2: An 8-processor computer connected via a $POPS(4, 2)$ network

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. [9] 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 message 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, 9]. A one-to-all broadcast can also be done in one slot [6, 9]. 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, 9] 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 [4]. Matrix multiplication and data routing algorithms for the POPS topology were developed in [12].

We use two numbering schemes for the processors of a $POPS(d, g)$. In the first numbering scheme the processors are numbered 0 through $n-1$ with processor i ($p(i)$) being the i mod 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.

In this paper we begin, in Section 2, by showing how a POPS computer can simulate SIMD hypercube and mesh computers. The simulation may be used to run hypercube and mesh algorithms on a POPS computer. The simulated versions of the best hypercube and mesh algorithms then become a benchmark against which we can compare the performance of any algorithms that are developed specifically for the POPS computer. In Sections 3 and 4 we develop custom algorithms for several of the fundamental arithmetic and data movement operations identified in [11]. As noted in [11] these fundamental operations are the building blocks for parallel algorithms for many applications.

2 Simulations

2.1 SIMD Hypercube Simulation

Let $i^{(b)}$ be the number whose binary representation differs from that of i only in bit b. The primitive communication in an $n = 2^D$ processor SIMD hypercube is for every processor i, $i \leq 0 < n$, to send data to processor $i^{(b)}$, for some fixed b, $0 \leq b < D$. This communication can be simulated by a $POPS(d, g)$ ($dg = n$) computer using $2\lceil d/g \rceil$ moves. So every n processor SIMD hypercube algorithm can be run on an n processor $POPS(d, g)$ with a slowdown of a factor of at most $2\lceil d/g \rceil$.

Theorem 1 *An n processor $POPS(d, g)$ can simulate every move of an n processor SIMD hypercube using 1 slot when $d = 1$ and $2\lceil d/g \rceil$ slots when $d > 1$.*

Proof: First consider the case $d = 1$. A $POPS(1, g)$ can perform any permutation π in 1 slot using the routing

$$p(i, 1) \rightarrow c(\pi(i), i) \rightarrow p(\pi(i), 1)$$

In particular, the data routing done in 1 hypercube move can be done in 1 slot.

Next consider the case $d > 1$. Processor i of the SIMD hypercube is mapped onto processor i of the $POPS(d, g)$. Let b be the bit along which the hypercube communication is to be done. That is, processor i of the hypercube is to send data to processor $i^{(b)}$ of the hypercube for all i. Let e and f be such that $ed + f = i$. That is $p(i)$ and $p(e, f)$ denote the same $POPS(d, g)$ processor.

First consider the case when $d = g = \sqrt{n} = 2^{D/2}$. When $b < D/2$, b is a bit of f and the communication can be done in two slots as below:

$$p(i) = p(e, f) \rightarrow c(f, e) \rightarrow p(f, e) \rightarrow c(e, f) \rightarrow p(e, f^{(b)}) = p(i^{(b)})$$

When $b \geq D/2$, b is a bit of e and the the communication is done in two slots as below:

$$p(i) = p(e, f) \rightarrow c(f, e) \rightarrow p(f, e) \rightarrow c(e^{(q)}, f) \rightarrow p(e^{(q)}, f) = p(i^{(b)})$$

The next case to consider is $d < \sqrt{n} < g$. When b is a bit of f we use the following 2 slot routing:

$$\begin{aligned} p(i) = p(e, f) &\rightarrow c(i \bmod g, e) && (1)\\ &\rightarrow p(i \bmod g, \lfloor i/g \rfloor) && (2)\\ &\rightarrow c(e, i \bmod g) && (3)\\ &\rightarrow p(e, f^{(b)}) = p(i^{(b)}) && (4) \end{aligned}$$

To establish the correctness of this routing we note that data that start in two different processors $i_1 = e_1 d + f_1$ and $i_2 = e_2 d + f_2$ use different couplers in slot 1 (i.e., for the source node to coupler routing of line 1). To see this observe that when $e_1 \neq e_2$ the slot 1 couplers differ in their second index. When $e_1 = e_2$, i_1 and i_2 differ by at most $d - 1 < g$ (and at least 1). So $i_1 \bmod g \neq i_2 \bmod g$, and the slot 1 couplers differ in their first index. Since $i = \lfloor i/g \rfloor g + i \bmod g$, data that originate in different processors are routed to different destination processors of line (2). For the slot 2 couplers (line (3)) we see that if $e_1 \neq e_2$, these couplers differ in their first index. When $e_1 = e_2$, i_1 and i_2 differ by at most $d - 1 < g$ (and at least 1). So the couplers differ in their second index.

When b is a bit of e the following 2 slot routing may be used:

$$p(i) = p(e, f) \rightarrow c(i \bmod g, e) \rightarrow p(i \bmod g, \lfloor i/g \rfloor) \rightarrow c(e^{(b)}, i \bmod g) \rightarrow p(e^{(b)}, f) = p(i^{(b)})$$

The correctness proof for this routing is similar to that for the case when b is a bit of f. Notice that the 2 slot routing for the case $d < \sqrt{n} < g$ works when $d = g = \sqrt{n}$ also.

The final case to consider is $d > \sqrt{n} > g$. This case is handled using $\lceil d/g \rceil$ passes of the simulation strategy for the case $d < \sqrt{n} < g$. Notice that when the 2 slot routing of lines 1–4 is used up to $\lceil d/g \rceil$ source processors from the same group e attempt to route data to the same coupler in lines 1 and 3. This conflict is resolved by dividing the processors in each group into $\lceil d/g \rceil$ subgroups and doing the routing for each subgroup in a different pass. ■

A SIMD hypercube algorithm may restrict a particular move along bit b to a subset of the processors. When the restriction is such that only one processor of each $POPS(d, g)$, $d > 1$, group is to send data, the simulation of Theorem 1 is not optimal. Suppose that a hypercube routing step is restricted in this way. When b is a bit of f, the data transfer along bit b may be accomplished using a single slot using the following routing:

$$p(i) = p(e, f) \rightarrow c(e, e) \rightarrow p(e, f^{(b)}) = p(i^{(b)})$$

Since for each $POPS(d, g)$ group e only one processor $p(e, f)$ is active, there is no coupler conflict in the above route. When b is a bit of e, the data transfer along bit b may be accomplished using a single slot as below:

$$p(i) = p(e, f) \rightarrow c(e^{(b)}, e) \rightarrow p(e^{(b)}, f) = p(i^{(b)})$$

Most SIMD hypercube data transfer steps along bit b will, however, require two or more processors of the same $POPS(d, g)$ group to transmit data. In this case the data transfer cannot be accomplished

in a single slot and the 2 slot simulation of Theorem 1 is optimal. To see this observe that when b is a bit of f the only single slot routes for data in $p(e_1, f_1)$ and $p(e_1, f_2)$, $f_1 \neq f_2$, are:

$$p(e_1, f_1) \rightarrow c(e_1, e_1) \rightarrow p(e_1, f_1^{(b)})$$

and

$$p(e_1, f_2) \rightarrow c(e_1, e_1) \rightarrow p(e_1, f_2^{(b)})$$

These routes have a conflict because they use the same coupler. So no single slot routing is possible. When b is a bit of e, the only single slot routes for the two data are:

$$p(e_1, f_1) \rightarrow c(e_1^{(b)}, e_1) \rightarrow p(e_1^{(b)}, f_1)$$

and

$$p(e_1, f_2) \rightarrow c(e_1^{(b)}, e_1) \rightarrow p(e_1^{(b)}, f_2)$$

These routes use the same coupler and so are in conflict.

2.2 SIMD Mesh Simulation

In an $N \times N$ SIMD mesh data may be moved one processor right/left/up/down along the rows/columns of the mesh in one step. The direction of data movement in a single step is the same for all data. In a mesh with wraparound, a rightward move by 1 causes data in the rightmost processor of each row to reach the leftmost processor of the row (left, up, and down moves handle end elements similarly). A communication step of a mesh with wraparound (and hence of a mesh with no wraparound) can be simulated by a $POPS(d, g)$ ($dg = N^2$ and either d divides N or g divides N) computer using $2\lceil d/g \rceil$ slots. So every n processor SIMD mesh algorithm can be run on an n processor $POPS(d, g)$ with a slowdown of a factor of at most $2\lceil d/g \rceil$.

Theorem 2 *An $n = N^2$ processor $POPS(d, g)$ (d or g divides N) can simulate every move of an $N \times N$ processor SIMD mesh with wraparound using 1 slot when $d = 1$ and $2\lceil d/g \rceil$ slots when $d > 1$.*

Proof: The proof for the case $d = 1$ is the same as that for the simulation of a hypercube move. So consider the case $d > 1$. Processor (i, j) of the mesh is mapped onto processor $p(e, f)$ of the $POPS(d, g)$, where $ed + f = iN + j$. As in the case of the proof of Theorem 1, we consider three cases. The first case has $d = g = N$. When $d = g = N$, a rightward row move by 1 is accomplished using the 2 slot routing:

$$p(e, f) \rightarrow c(f, e) \rightarrow p(f, e) \rightarrow c(e, f) \rightarrow p(e, (f + 1) \bmod d)$$

A downward column move by 1 is done by the following 2 slot routing:

$$p(e,f) \rightarrow c(f,e) \rightarrow p(f,e) \rightarrow c((e+1) \bmod g, f) \rightarrow p((e+1) \bmod g, f)$$

Leftward row moves and upward column moves are done similarly.

When $d < N < g$ each row of the mesh occupies more than one group of the $POPS(d,g)$. Since d divides N, each row is housed in an integral number N/d of groups. A rightward shift by 1 along rows of the mesh requires data in processor $p(e,f)$ of the $POPS(d,g)$ to be routed to $p(e',f')$, where

$$e' = \begin{cases} e & \text{if } f \neq d-1 \\ e+1 & \text{if } f = d-1 \text{ and } (e+1) \bmod N/d \neq 0 \\ e+1-N/d & \text{if } f = d-1 \text{ and } (e+1) \bmod N/d = 0 \end{cases}$$

and

$$f' = (f+1) \bmod d$$

Let $r = ed + f$. The following 2 slot routing moves data from $p(e,f)$ to $p(e',f')$ for all e and f.

$$\begin{aligned} p(e,f) &\rightarrow c(r \bmod g, e) && (5) \\ &\rightarrow p(r \bmod g, \lfloor r/g \rfloor) && (6) \\ &\rightarrow c(e', r \bmod g) && (7) \\ &\rightarrow p(e',f') && (8) \end{aligned}$$

The proof that different source processors use different slot 1 couplers (i.e., in line (5)) and different slot 1 destination processors (line (6)) is the same as for the corresponding lines of the proof of Theorem 1. Suppose that data originates in two different processors $p(e_1,f_1)$ and $p(e_2,f_2)$ in line (5). If $e'_1 \neq e'_2$ there is no coupler conflict in line (7). If $e'_1 = e'_2$, then $p(e_1,f_1)$ and $p(e_2,f_2)$ represent the same row of the mesh. Hence, $0 < |r_1 - r_2| < N < g$. So $r_1 \bmod g \neq r_2 \bmod g$ and there is no coupler conflict in line (7).

For a downward column move $e' = (e+N/d) \bmod g$ and $f' = f$. A downward move may also be done using the routing of lines 5–8. Leftward row moves and upward column moves are similarly dome in 2 slots per move.

As was the case in the proof of Theorem 1, the case $d > N > g$ is handled using $\lceil d/g \rceil$ passes of the routing of lines 5–8. ■

The comments made following the proof of Theorem 1 regarding the optimality of the simulation apply to our simulation of a mesh also. When $d \geq 2\sqrt{n}$, a similarly optimal simulation of a 2D-mesh may be obtained using the embedding of a tori described in [2, 6].

3 Arithmetic Operations

We explicitly consider only the case $d > 1$. When $d = 1$, the arithmetic operations conidered in this section are best done by simulating the corresponding hypercube algorithms using Theorem 1. Further, for simplicity in exposition we assume that d and g are powers of 2. When one or both of d and g are not powers of 2, the algorithms we develop may be used by conceptually rounding d and/or g to the next power of 2 and ignoring operations that involve processors that do not exist.

3.1 Data Sum

In this operation we are to sum the data values in each processor; the sum is to be left in processor $p(0,0)$. Although we can do this operation in $2\lceil d/g \rceil \log_2 n$ slots by simulating the hypercube data sum algorithm of [11] (see Theorem 1), a faster custom algorithm is possible. In fact, the data sum operation can be done in half as many slots, that is in $\lceil d/g \rceil \log_2 n$ slots.

First consider the case $d = g = \sqrt{n}$. We show how to reduce the problem of summing in a $POPS(\sqrt{n}, \sqrt{n})$ to one of summing in a $POPS(\sqrt{n}/2, \sqrt{n/2})$ using 2 slots. By applying this reduction process $(\log_2 n)/2$ times the problem is reduced to summing in a $POPS(1,1)$, which requires no work. So the total number of slots taken for the data sum operation is $\log_2 n$.

Figure 3(a) shows a symbolic diagram of a $POPS(m,m)$. In this figure the processors are layed out in a 2-dimensional array with processors in the same group defining a row of the array. The top-left corner of the array is $p(0,0)$. Initially, the data to be summed is distributed over the m^2 processors, 1 datum per processor. We shall first fold the data into $m^2/2$ processors so that each processor has 2 data and then each processor will add its 2 data. The folding into $m^2/2$ processors is done by pairing processors. Every processor below the fold line (the fold line is the antidiagonal of the processor array; $p(i,j)$ is on the fold line iff $i + j = m - 1$) of Figure 3(a) is paired with a processor above this fold line. To determine a pair of processors, simply fold at the fold line and pair processors that line up. To pair processors that lie on the fold line, superpose processors on the bottom half of the fold line and those on the top half by sliding the bottom half up by $m/2$ positions as shown in Figure 3(b). The folding and sliding substeps defined result in $m^2/2$ processor pairs. Formally, $p(i,j)$ and $p(m-1-j, m-1-i)$ form a processor pair when $i + j \neq m - 1$. When $i + j = m - 1$, $p(i,j)$ is an antidiagonal processor and the processor pair is $(p(i,j), p(i - m/2, j + m/2))$ (this is equivalent to $(p(i, m-1-i), p(i - m/2, 3m/2 - 1 - i))$).

The stated reduction to $m^2/2$ processors is accomplished in 1 slot by the following routing. For all i

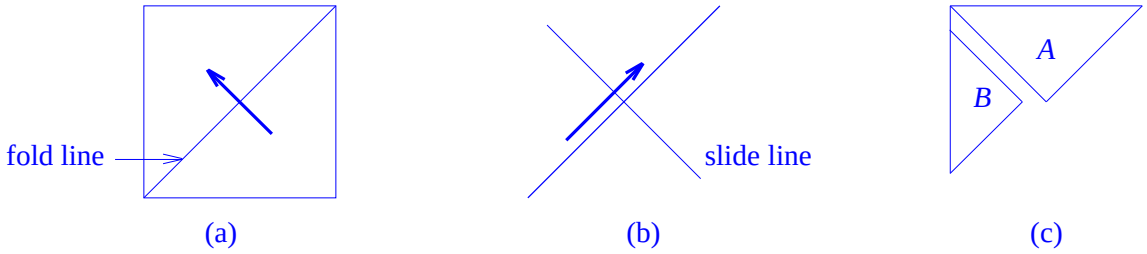

Figure 3: First step of data sum reduction process

and j such that $i + j > m - 1$ (i.e., for all processors below the antidiagonal) route as below:

$$p(i, j) \rightarrow c(m - 1 - j, i) \rightarrow p(m - 1 - j, m - 1 - i) \tag{9}$$

For $i \geq n/2$ route as below:

$$p(i, m - 1 - i) \rightarrow c(i - m/2, i) \rightarrow p(i - m/2, 3m/2 - 1 - i) \tag{10}$$

You may easily verify that no two processors below the fold line of Figure 3(a) use the same coupler in line (9) and that no two processors below the slide line of Figure 3(b) use the same coupler in line (10). To see that there is no coupler conflict between lines 9 and 10, note that source processors in different rows of Figure 3(a) use couplers that differ in their second index. For source processor $p(i, m-1-i)$ to have a coupler conflict with source processor $p(i, j)$ that is below the fold point, we must have $m-1-j = i-m/2$. In other words, $i + j = m/2 - 1$. This is not possible because $i + j > m - 1$ since $p(i, j)$ is below the fold line.

Following the above routing each destination processor adds the two data it has. We now have $m^2/2$ data to sum, and these data are distributed one per processor in the processors that lie in the triangles A and B of Figure 3(c). The next step is to reduce the number of data to be summed to $m^2/4$ and have these data distributed one per processor in the $m/2 \times m/2$ top-left subarray. To accomplish this objective divide the triangles A and B into two halves as shown in Figure 4(a). The triangles A_1 and B_1 define the $m/2 \times m/2$ subarray in which the $m^2/4$ data to be summed are to be left.

Each processor of A_2 is paired with a processor of A_1 by rotating A_2 90 degrees clockwise (see Figure 4(b), A_2' is the result of rotating A_1 90 degrees clockwise) and then sliding the rotated triangle left by $m/2$ positions. Formally, $p(i, j)$ such that $i < n/2$, $j \geq n/2$, and $i + j < m$ is paired with $p(j - m/2, m/2 - 1 - i)$. Data from $p(i, j)$, such that $i < n/2$, $j \geq n/2$, and $i + j < m$ may be routed to its paired processor using the one slot routing:

$$p(i, j) \rightarrow c(j - m/2, i) \rightarrow p(j - m/2, m/2 - 1 - i) \tag{11}$$

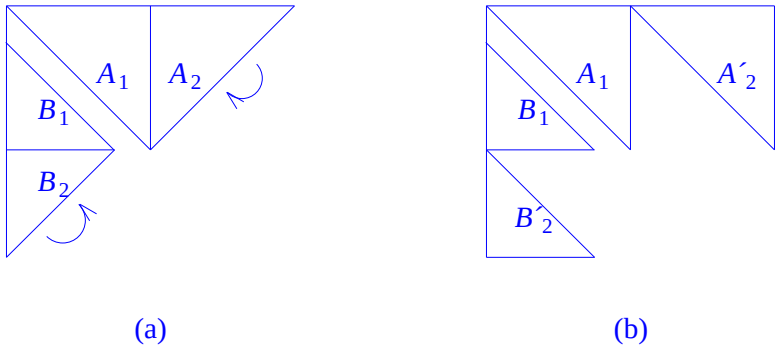

Figure 4: Second step of data sum reduction process

You may verify that there are no coupler conflicts in the routing of line (11).

Each processor of B_2 is paired with a processor of B_1 by rotating B_2 90 degrees counterclockwise (see Figure 4(b)) and then sliding the rotated triangle up by $m/2-1$ positions. Formally, $p(i,j)$ such that $i \geq m/2$, $j < m/2-1$, and $i+j < m-1$ is paired with processor $p(m/2-j-1, i-m/2)$. Data from $p(i,j)$ such that $i \geq m/2$, $j < m/2-1$, and $i+j < m-1$ may be routed to its paired processor using the one slot routing:

$$p(i,j) \rightarrow c(m/2-j-1, i) \rightarrow p(m/2-j-1, i-m/2) \tag{12}$$

Again, it is easy to see that this routing has no coupler conflicts. Further, no coupler conflicts arise when the routings of lines 11 and 12 are done in the same slot because $i < m/2$ for the couplers used in line (11) and $i \geq m/2$ for the couplers used in line (12). Therefore, the couplers used in lines 11 and 12 differ in their second index.

We now turn our attention to the case when $d < \sqrt{n} < g$. We now view the $POPS(d,g)$ as a $g \times d$ array with each row of the array representing a group of processors. $\log_2(g/d)$ passes of a reduction step that halves the number of groups are first made to reduce the summing problem to one that involves only the top-left $d \times d$ subarray of processors; each processor in this subarray has one datum that is to be summed. Then the $m \times m$ subarray algorithm described above is used to complete the summing using an additional $2\log_2 m = 2\log_2 d$ slots. Since each pass of the group halving reduction step takes 1 slot, the total number of slots is $\log_2(g/d) + 2\log_2 d = \log_2 g + \log_2 d = \log_2 gd = \log_2 n$.

Suppose we have m groups. In a group halving step, each processor in a group i, $i \geq m/2$ is paired with a processor in a group i, $i < m/2$. Specifically, $p(i,j)$ for $i \geq m/2$ is paired with $p(\lfloor i/d \rfloor d - m/2 + j, i \bmod d)$. Data from $p(i,j)$ for $i \geq m/2$ is routed to its paired processor in 1 slot using the routing:

$$p(i,j) \rightarrow c(\lfloor i/d \rfloor d - m/2 + j, i) \rightarrow p(\lfloor i/d \rfloor d - m/2 + j, i \bmod d)$$

This routing is easily seen to have no coupler conflict. Following the above routing, each processor in the top-left $m/2 \times d$ subarray has 2 data. These data are added and we are left with a data sum problem that involves only the top-left $m/2 \times d$ subarray.

The final case to consider is when $d > \sqrt{n} > g$. This case is handled by making g/d passes of the $m \times m$ algorithm. The details are similar to those used in the proof of Theorem 1.

3.2 Prefix Sum

In this operation each processor $p(i,j)$ of the $POPS(d,g)$ begins with a value $a(i,j)$. $p(i,j)$ is to compute the prefix sum $prefixSum(i,j) = \sum_{k<i} \sum_{0 \le q < d} a(k,q) + \sum_{q \le j} a(i,q)$. From Theorem 1 and the hypercube prefix sum algorithm of [11] it follows that a $POPS(d,g)$ can compute prefix sums in $2\lceil d/g \rceil \log_2 n$ slots. We develop a customized $POPS(d,g)$ algorithm that computes the prefix sums in $3 + \log_2 n + \log_2 d$ slots when $1 < d \le g$ and $2d/g(1 + \log_2 g) + \log_2 d + 1$ slots when $d > g$.

When $d = g = \sqrt{n}$ the steps in our prefix sum algorithm are:

Step 1: Compute the group prefix sums $groupPrefixSum(i,j) = \sum_{0 \le k \le j} a(i,k)$ for $0 \le i < g$ and $0 \le j < d$.

Step 2: Compute the prefix sums of $groupPrefixSum(i, d-1)$, $0 \le i < g$.
Let $s(i,j) = \sum_{0 \le k < i} groupPrefixSum(k, d-1)$.

Step 3: Compute $prefixSum(i,j) = groupPrefixSum(i,j) + s(i,j)$, $0 \le i < g$, $0 \le j < d$.

Assume that $n > 1$. The group prefix sums are computed in $2 + 2\log_2 g$ slots as follows. First transpose the $a(i,j)$s using the routing:

$$p(i,j).a \rightarrow c(j,i) \rightarrow p(j,i).b$$

The notation $p(i,j).a$ refers to the a value of $p(i,j)$. Now $p(i,j)$ recursively computes $t(i,j) = \sum_{0 \le k \le i} b(k,j) = \sum_{0 \le k \le i} a(j,k) = groupPrefixSum(j,i)$ in the following way. The initial partition size (i.e., number of $POPS$ groups, each group has d processors) is $m = g$. When $m = 1$, $t(i,j) = b(i,j)$. When $m > 1$ divide the partition (say partition A) into two equal-sized partitions B and C. B and C each have $m/2$ $POPS$ groups of d processors each. The group indexes in B are less than those in C. Compute $t(i,j)$ in each subpartition B and C as though there were no other (sub)partitions. The computed $t(i,j)$ values for the processors in B are accurate even with respect to the partition A. The $t(i,j)$ values computed for C need to be increased by $t(last, j)$, where $last$ is the highest indexed group in B. The $t(last, j)$ values

may be routed in 2 slots to the $p(i,j)$s of C using the routing:

$$p(last,j) \rightarrow c(j,last) \rightarrow p(j,last) \rightarrow c(*,j) \rightarrow p(*,j)$$

where $*$ refers to all group indexes in C. Once the processors in C receive the $t(last,j)$ values, they add the received value to their current $t(i,j)$ value.

Following the recursive computation of $t(i,j)$, the $t(i,j)$ values are transposed using the single slot routing

$$p(i,j) \rightarrow c(j,i) \rightarrow p(j,i)$$

and now $p(i,j)$ has the value of $groupPrefixSum(i,j)$.

Step 2 is done in $1+\log_2 g$ slots. First, for all i, $p(i,d-1)$ computes $s(i,d-1)$ recursively and then broadcasts $s(i,d-1)$ to all processors in group i. The broadcast to all processors within the group takes 1 slot using the routing

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

where $*$ denotes all indexes in the range 0 through $d-1$.

For the recursive computation of $s(d-1)$, start with a single partition of size $m=g$. If $m=1$, set $s(i,d-1) = groupPrefixSum(i,d-1)$. If $m>1$ divide a size m partition A into two size $m/2$ partitions B and C (as above). Compute $s(i,d-1)$ for the processors of B and C independently. For partition A the computed $s(i,d-1)$ values of B are not changed. For partition C we add $s(last,d-1)$ to all $s(i,d-1)$ in C. For this purpose use the routing

$$p(last,d-1) \rightarrow c(*,last) \rightarrow p(*,last)$$

where $*$ refers to all group indexes that are in C. When the recursion terminates, subtract $groupPrefixSum(i,d-1)$ from $s(i,d-1)$ for all i.

Step 2 requires zero slots. The total number of slots required by our algorithm is $2+2\log_2 g+1+\log_2 g = 3+3\log_2 g = 3+\log_2 n+\log_2 d$

Now consider the case $1<d<\sqrt{n}<g$. Step 1 may be done in $2+2\log_2 d$ slots by partitioning the processors of the $POPS(d,g)$ into g/d partitions, each comprising d groups of size d. In each of the g/d partitions, run the step 1 algorithm for case $d=g$. For step 2, $p(i,d-1)$ first computes $s(i,d-1)$ and then broadcasts $s(i,d-1)$ to the remaining processors of the group. The number of $s(i,d-1)$s that are to be computed is $m=g$. When $m>d$, partition into two halves B and C and compute for each half independently. The i values in B are less than those in C. Following the computation of the s values in

each half, update the computed values in C by adding $s(last, d-1)$, where $last$ is the maximum group index in B. This update takes 1 slot. When $m = d$, use the step 2 algorithm for the case $g = d$ to compute $s(i, d-1)$ in $\log_2 d$ slots. The total number of slots needed to compute $s(i, d-1)$, $0 \leq i < g$ is $\log_2(g/d) + \log_2 d = \log_2 g$. An additional slot is used for the intragroup broadcast of $s(i, d-1)$ when $d > 1$. So steps 1 and 2 can be done in $3 + 2\log_2 d + \log_2 g = 3 + \log_2 n + \log_2 d$ slots.

When $d > \sqrt{n} > g$, step 1 may be done in $2d/g(1 + \log_2 g)$ slots by making d/g passes (the passes are numbered 1 through d/g) of a modified version of the algorithm for the case $d = g$. Each pass computes the group prefix sum for g of the as in a group. The algorithm for pass q differs from the step 1 algorithm for the case $d = g$ only in that the final transpose of t not only transposes t but also sends $t(g-1, j)$ to processor $p(j, qg)$ that will be active in the next pass (if any). Processor $p(j, qg)$ adds the received t value to its a value before the start of the next pass. So, for example, at the end of pass 1 $p(g-1, j)$ sends $t(g-1, j) = groupPrefixSum(j, g-1)$ to $p(j, g-1)$ and $p(j, g)$, processor $p(j, g)$ adds the received t value to its a value before the start of pass 2; at the end of pass 2 $p(g-1, j)$ sends $t(g-1, j) = groupPrefixSum(j, 2g-1)$ to $p(j, 2g-1)$ and $p(j, 2g)$, processor $p(j, 2g)$ adds the received t value to its a value before the start of pass 3. Step 2 is done the same as when $d = g$.

Note that the ranking operation [11] in which each processor has a boolean value $selected(i, j)$ and we are to determine, for each processor, the number of lexicographically smaller processors with $selected(i, j) = true$ can be done using a slight modification of the preceding prefix sum algorithm.

3.3 Consecutive Sum

This operation was defined by Kumar and Krishnan [7]. Each processor $p(i, j)$ of the $POPS(d, g)$ begins with an array $p(i, j).X[*]$ of $M < d$ (M divides d) numbers; the processors in each group are divided into subgroups of size M with $p(i, j)$ being in the $\lceil j/M \rceil$th subgroup of group i; and $p(i, j)$ is to compute the sum $s(i, j)$ of the $X[q]$s in its subgroup, where $q = j \bmod M$.

The hypercube algorithm for the consecutive sum operation makes M hypercube moves [11]. Using Theorem 1 we may simulate this hypercube algorithm to obtain a $POPS(d, g)$ algorithm for the consecutive sum operation. The simulation takes $2\lceil d/g \rceil M$ slots. We develop a custom algorithm that performs the consecutive sum operation in $\lceil d/g \rceil M$ slots when $M \leq g$ and in $\lceil d/g \rceil (g+1)$ slots when $M > g$.

First consider the case $d = \sqrt{n} = g$. We describe two algorithms for this case. The first algorithm is quite simple but requires each processor to transmit an entire X array in each slot. The second algorithm transmits an entire array in just 1 slot and singleton elements in each of the remaining slots. When M is large a single slot transmission of an array of size M is not possible and we need to change our analysis

taking into account the network bandwidth B. When this bandwidth is accounted for, the first algorithm takes M^2/B slots while the second take $M/B + M - 1$ slots. The hypercube simulation for the case $d = g$ still takes $2M$ slots.

In the first $d = g$ algorithm each processor $p(i,j)$ begins by setting $s(i,j) = p(i,j).X[q]$, where $q = j \bmod M$. Then each processor broadcasts its X arrays to all processors that are in the same subgroup. When processor $p(i,j)$ receives an X array it adds the received $X[q]$ value to $s(i,j)$. The broadcast of the X arrays is done using the routing:

$$p(i,j) \rightarrow c(j,i) \rightarrow p(j,i) \rightarrow c(i,j) \rightarrow p(i,*)$$

where $*$ denotes all processors that are in the same subgroup as $p(i,j)$. The broadcast takes $M-1$ slots as each processor is to receive $M-1$ arrays and may receive only one array per slot; the transmission $p(j,i) \rightarrow c(i,j) \rightarrow p(i,*)$ is repeated $M-1$ times to allow each receiving processor enough time to receive all $M-1$ arrays that it needs.

Our second algorithm uses the following 2 steps:

Step 1: Transpose the X arrays so that $p(i,j).Y[k] = p(j,i).X[k]$ for all i, j, and k. For this purpose use the routing $p(i,j) \rightarrow c(j,i) \rightarrow p(j,i)$.

Step 2: $s(i,j)$ is computed by initiating a token S in $p(r + (j+1) \bmod M, i)$, where $r = \lfloor j/M \rfloor * M$. The initial value of this token is $p(r+(j+1) \bmod M, i).Y[q] = p(i, r+(j+1) \bmod M).X[q]$, where $q = j \bmod M$. Notice that $p(i, r+(j+1) \bmod M).X[q]$ is one of the terms in $s(i,j)$. Token S will eventually be in processor $p(i,j) = p(i, r+q)$. The path taken by this token is:

$$\begin{aligned}
p(r+(j+1) \bmod M, i) &\rightarrow c(i, r+(j+1) \bmod M) && (13)\\
&\rightarrow p(i, r+(j+2) \bmod M) && (14)\\
&\rightarrow c(r+(j+3) \bmod M, i) && (15)\\
&\rightarrow p(r+(j+3) \bmod M, i) && (16)\\
&\rightarrow c(i, r+(j+3) \bmod M) && (17)\\
&\rightarrow p(i, r+(j+4) \bmod M) && (18)\\
&\rightarrow c(r+(j+5) \bmod M, i) && (19)\\
&\rightarrow p(r+(j+5) \bmod M, i) && (20)
\end{aligned}$$

$\rightarrow$.

$\rightarrow$.

$\rightarrow$.

$\rightarrow$ $c(i, r + (q-1) \bmod M)$ (21)

$\rightarrow$ $p(i, r+q)$ (22)

(23)

When token S reaches a processor $p(i,t)$, $p(i,t).X[q]$ is added to the token, and when a processor $p(t,i)$ is reached, $p(t,i).Y[q] = p(i,t).X[q]$ is added. It is easy to see that after $M-1$ slots token S will be in $p(i, r+q) = p(i,j)$ and would have computed the desired consecutive sum. You may verify that there are no coupler conflicts in lines 13, 15, 17, $\cdots$ or destination processor conflicts in lines 14, 16, 18, $\cdots$ of the given routing.

When $d < \sqrt{n} < g$, we may partition the $POPS(d,g)$ into g/d $POPS(d,d)$s and run either of the above algorithms for the case $d = g$ in each of these $POPS(d,d)$s. The number of slots required remains M.

When $d > \sqrt{n} > g$ the $POPS(d,g)$ is partitioned into d/g $POPS(g,g)$s. We consider the two cases (a) $M \leq g$ and (b) $M > g$. Case (a) is handled in d/g passes. Each pass uses either of the $d = g$ algorithms on one of these partitions. The number of slots required is $(d/g)M$. For case (b) we get a different slot count depending on which of the two $d = g$ algorithms we use. When the first algorithm is used, we make d/g passes working with a $POPS(g,g)$ in each pass. However, in the broadcast stage rather than broadcast only to the g processors in a group of the $POPS(g,g)$, we broadcast to all $M-1$ processors that need the X array being broadcast. This broadcast takes g slots (rather than $g-1$) because $M - g > 0$ processors need all of the g X arrays that are being broadcast. The total slot count is therefore, $(d/g)(g+1)$.

When the second $d = g$ algorithm is used, cluster the $POPS(g,g)$s so that each cluster is a $POPS(g,M)$ and so that each cluster contains all the data needed to compute the consecutive sum for its processors. Each processor in a $POPS(g,g)$ computes the contribution to the consecutive sum that is to be computed by the corresponding processors in the M/g $POPS(g,g)$s that make up its cluster. For each $POPS(g,g)$, this is done by running step 1 of the $d = g$ algorithm once and running step 2 M/g times. The number of slots required to do this for a single cluster is $(1 + M/g * (g-1)) * M/g = (1 + M - M/g)M/g$. Then the computed contributions are to be routed to the proper processors for addition. For each $POPS(g,g)$

this routing is done by first transposing the data that is to be routed in 1 slot and then using 1 slot per destination $POPS(g,g)$ for transmission. The total number of slots per cluster is M^2/g. So the computation for each cluster takes $(M+1)M/g$ slots. Since the number of clusters is d/M, the complete consecutive sum computation takes $(d/g)(M+1) < (d/g)(g+1)$ slots. So the second algorithm is inferior, for this case, when the bandwidth is such as to permit the transmission of an entire X array in a single slot.

3.4 Adjacent Sum

The initial configuration for the adjacent sum operation [7] is the same as for the consecutive sum operation. Now, however, processor $p(i,j)$ is to compute $s(i,j) = \sum_{0 \le k < M} p(i,(j+k) \bmod d).X[k]$. The hypercube algorithm for this task [11] may be simulated in $2\lceil d/g \rceil (2M + \log_2(n/M))$ slots. By recognizing that the two data transmissions done in each iteration of the **for** loop of the algorithm of [11] can be done as a single transmission in a $POPS$ (because of the higher optical bandwidth assumption), the number of slots is reduced to $2\lceil d/g \rceil (M + \log_2(n/M))$. A simple $2\lceil d/(1+g) \rceil (M-1)$ slot algorithm results if we simply shift the X arrays within each group circularly left by 1 $M-1$ times, following each shift the processors consume the needed value from the newly received X array. Each circular shift is done in $2\lceil d/(1+g) \rceil$ slots using the multiple intragroup permutation algorithm of [12].

Our first (and simpler) consecutive sum algorithm (Section 3.3) is easily adapted to the adjacent sum problem. In fact the algorithm for the cases $d = g$ and $d < g$ may be used with virtually no change. The number of slots for each case remains M. The case $d > g$ is handled in d/g passes, and in each pass each of the the g processors in a group of a $POPS(g,g)$ broadcasts its X array to the $M-1$ processors that need an X value from this array. This broadcast takes $\min\{M,g\}$ slots per $POPS(g,g)$. The total number of slots is, therefore, $(d/g)\min\{M,g\}$.

4 Data Movement Operations

$POPS$ algorithms for some of the fundamental data movement operations of [11] have been developed earlier. For example, data broadcasting algorithms are developed in [6, 9] and algorithms for intragroup permutations (this includes data shift within groups), the frequently arising permutations of [8] and the bit-permute-complement class of permutations of [10] are developed in [12]. In this section we develop $POPS$ algorithms for the following data movement operations: concentrate, distribute, and generalize.

4.1 Concentrate

For this operation some of the $POPS(d, g)$ processors are labeled as *selected* processors. Each selected proccesor has a data D and a rank r associated with it (the rank of a selected processor $p(i, j)$ is the number of selected processors that are lexicographically smaller than $p(i, j)$; this rank may be computed by assigning a value $s = 1$ to each selected processor and $s = 0$ to each processor that is not selected, and then computing the prefix sums of the s values). Data from each selected processor $p(i, j)$ is to be routed to $p(r) = p(\lfloor r/d \rfloor, r \bmod d)$. The hypercube data concentration algorithm of [11] makes $\log_2 n$ moves. Simulating this algorithm on a $POPS(d, g)$ gives a $POPS(d, g)$ data concentration algorithm that takes $2\lceil d/g \rceil \log_2 n$ slots. Data concentration can be done in $2\lceil d/g \rceil$ slots using a customized $POPS$ algorithm.

When $d = \sqrt{n} = g$, we use the following 2 slot algorithm (here $p(i, j)$ denotes a selected processor):

$$
\begin{aligned}
p(i, j) &\rightarrow c(r \bmod d, i) && (24)\\
&\rightarrow p(r \bmod d, i) && (25)\\
&\rightarrow c(\lfloor r/d \rfloor, r \bmod d) && (26)\\
&\rightarrow p(\lfloor r/d \rfloor, r \bmod d) && (27)
\end{aligned}
$$

There is no coupler conflict in line (24) because the ranks of two selected processors that are in the same row i may differ by at most $d - 1$. Since there is no coupler conflict in line (24), there is no destination processor conflict in line (25) (destination processor index is the same as the coupler index). Further, there is no coupler conflict in line (26) because the ranks of selected processors are distinct. So if $r_1 \neq r_2$ and $r_1 \bmod d = r_2 \bmod d$, then $\lfloor r_1/d \rfloor \neq \lfloor r_2/d \rfloor$.

When $d < \sqrt{n} < g$ the following 2 slot routing may be used:

$$
\begin{aligned}
p(i, j) &\rightarrow c(r \bmod g, i) && (28)\\
&\rightarrow p(r \bmod g, \lfloor r/g \rfloor) && (29)\\
&\rightarrow c(\lfloor r/d \rfloor, r \bmod g) && (30)\\
&\rightarrow p(\lfloor r/d \rfloor, r \bmod d) && (31)
\end{aligned}
$$

There is no coupler conflict in line (28) because two selected processors that are in the same $POPS$ group have a rank difference that is at most $d - 1$ and $g > d$. There is no destination processor conflict

in line 29 because all ranks are distinct. So if $r_1 \neq r_2$ and $r_1 \bmod g = r_2 \bmod g$, then $\lfloor r_1/g \rfloor \neq \lfloor r_2/g \rfloor$. There is no coupler conflict in line (30) because if $r_1 \neq r_2$ and $r_1 \bmod g = r_2 \bmod g$, then $|r_1 - r_2| \geq g > d$. So the destination groups $\lfloor r_1/d \rfloor$ and $\lfloor r_2/d \rfloor$ must be different.

The case $d > \sqrt{n} > g$ is handled by making d/g passes of the $d = g$ algorithm. For this purpose, replace each occurrence of mod d in lines 24–26 with mod g. In one pass, work on a $g \times g$ block of processors of the $POPS(d, g)$. The total number of slots required is $2d/g$.

4.2 Distribute

This is the inverse of a concentrate. Data begins in $q \leq n$ processors $p(k), 0 \leq k < q$; each data has a destination processor $p(dest(i))$ such that $dest(0) < dest(1) < \cdots < dest(q-1)$; data initially in $p(i)$ is to be routed to processor $p(dest(i))$, $0 \leq i < q$. In most parallel architectures (e.g., hypercube and mesh) the distribute operation is implemented by running the concentrate algorithm backwards. In the case of the $POPS$ architecture it is not possible to run an algorithm backwards because of the asymmetry in the connection topology. For example, line (24) sends data from processor $p(i, j)$ to coupler $c(r \bmod d, i)$. However, it is not possible to send data directly from coupler $c(r \bmod d, i)$ to processor $p(i, j)$ (unless $i = r \bmod d$).

Since the hypercube algorithm for the distribute operation makes $\log_2 n$ moves, its $POPS$ simulation takes $2\lceil d/g \rceil \log_2 n$ slots. A customized $POPS$ algorithm can perform the distribute operation in $2\lceil d/g \rceil$ slots.

When $d = \sqrt{n} = g$ use the following 2 slot routing ($dest(i, j) = dest(id + j)$):

$$
\begin{aligned}
p(i,j) &\rightarrow c(j,i) && (32)\\
&\rightarrow p(j,i) && (33)\\
&\rightarrow c(\lfloor dest(i,j)/d \rfloor, j) && (34)\\
&\rightarrow p(\lfloor dest(i,j)/d \rfloor, dest(i,j) \bmod d) && (35)
\end{aligned}
$$

It is easy to see that there are no conflicts in lines 32, 33, and 35. For line (34) notice that when $i_1 \neq i_2$, $|dest(i_1, j) - dest(i_2, j)| \geq d$. Therefore, $\lfloor dest(i_1, j)/d \rfloor \neq \lfloor dest(i_2, j)/d \rfloor$.

When $d < \sqrt{n} < g$ use the following 2 slot routing:

$$
p(i,j) \rightarrow c((id + j) \bmod g, i) \qquad (36)
$$

$$\rightarrow \quad p((id+j) \bmod g, \lfloor (id+j)/g \rfloor) \tag{37}$$

$$\rightarrow \quad c(\lfloor dest(i,j)/d \rfloor, (id+j) \bmod g) \tag{38}$$

$$\rightarrow \quad p(\lfloor dest(i,j)/d \rfloor, dest(i,j) \bmod d) \tag{39}$$

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 line (36). For this observe that when $i_1 \neq i_2$ the 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 destination processors (see line (37)) 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 (38) observe that $p(i_1, j_1) \neq p(i_2, j_2)$ and $\lfloor dest(i_1, j_1)/d \rfloor = \lfloor dest(i_2, j_2)/d \rfloor$ imply that $0 < |i_1 d + j_1 - (i_2 d + j_2)| < d < g$. Therefore, $(i_1 d + j_1) \bmod g \neq (i_2 d + j_2) \bmod g$.

The case $d > \sqrt{n} > g$ is handled by making d/g passes of the $d = g$ algorithm. For this purpose, replace the last occurrence of j in each of the lines 32–34 with $j \bmod g$. In one pass, work on a $g \times g$ block of processors of the $POPS(d, g)$. The total number of slots required is $2d/g$.

4.3 Generalize

The initial configuration for the generalize operation is the same as that for the distribute operation. Let $dest(-1) = 0$. Data originally in $p(i)$ is to be routed to all processors $p(k)$ such that $dest(i-1) < k \leq dest(p(i))$, $0 \leq i < q$. The hypercube algorithm for this operation makes $\log_2 n$ moves. Therefore, we can obtain a $2\lceil d/g \rceil \log_2 n$ slot $POPS(d, g)$ algorithm for the generalize operation by simulating the corresponding hypercube algorithm. We develop a $4\lceil d/g \rceil$ slot custom algorithm to generalize on a $POPS(d, g)$.

When $d = \sqrt{n} = g$ we use the following 2 step algorithm:

Step 1: Shift the *dest* values right by 1 with a shift from the end of a group going to the first processor in the next group. The *dest* value received by $p(i, j)$ is called $lowDest(i, j)$. Set $lowDest(0, 0) = 0$. This shift is done in 2 slots using the following routing:

$$\begin{aligned} p(i,j) &\rightarrow c(j,i) \\ &\rightarrow p(j,i) \end{aligned}$$

$$\rightarrow \begin{cases} c(i,j) & j \neq d-1 \\ c(i+1,j) & i \neq g-1 \text{ and } j = d-1 \end{cases}$$

$$\rightarrow \begin{cases} p(i,j+1) & j \neq d-1 \\ p(i+1,0) & i \neq g-1 \text{ and } j = d-1 \end{cases}$$

You may verify that this routing has no conflicts.

Step 2: Now do the generalize using the routing:

$$p(i,j) \rightarrow c(j,i) \tag{40}$$

$$\rightarrow p(j,i) \tag{41}$$

$$\rightarrow c(\lfloor lowDest(i,j)/d \rfloor : \lfloor dest(i,j)/d \rfloor, j) \tag{42}$$

$$\rightarrow p(*,*) \tag{43}$$

where the notation $c(a:b,j)$ in line (42) refers to the coupler set $c(a,j)$, $c(a+1,j)$, $\cdots$, $c(b,j)$, and $p(*.*)$ in line (43) refers to all processors that are to receive the data initially in $p(i,j)$. It is easy to see that there are no conflicts in lines 40, 41, and 43. To see that there is no coupler conflict in line (42) observe that $dest(i_1,j) + d \leq lowDest(i_2,j)$ for $i_1 < i_2$. Therefore, $\lfloor dest(i_1,j)/d \rfloor < \lfloor lowDest(i_2,j)/d \rfloor$.

When $d < \sqrt{n} < g$ we use the same steps as used above. However, the routing formulas are more elaborate. For the shift we use the routing:

$$p(i,j) \rightarrow c((id+j) \bmod g, i) \tag{44}$$

$$\rightarrow p((id+j) \bmod g, \lfloor (id+j)/g \rfloor) \tag{45}$$

$$\rightarrow \begin{cases} c(i,(id+j) \bmod g) & j \neq d-1 \\ c(i+1,(id+j) \bmod g) & i \neq g-1 \text{ and } j = d-1 \end{cases} \tag{46}$$

$$\rightarrow \begin{cases} p(i,j+1) & j \neq d-1 \\ p(i+1,0) & i \neq g-1 \text{ and } j = d-1 \end{cases}$$

The routing of lines 44 and 45 is the same as that used in lines 36 and 37. So there are no conflicts in these lines. The proof that line (46) is conflict free is the same as that for line (44).

For step 2 we similarly replace the last occurrence of j in lines 40 through 42 with $(id+j) \bmod g$.

When $d > \sqrt{n} > g$ we first do the shift by making d/g passes of the shift algorithm for the case $d = g$. Then we simulate d/g passes of the step 2 routing.

5 Conclusion

We have shown that the $POPS$ computer is rather versatile; it can simulate mesh and hypercube computers that have the same number of processors with a slow down that is a factor of $2\lceil d/g \rceil$. However, it is possible to develop customized algorithms for the $POPS$ topology that exhibit a smaller slowdown. In fact, for some data routing problems, such as concentrate, distribute, and generalize the customized algorithms are asymptotically faster. For instance, each of these operations can be done in $O(1)$ slots on a $POPS(g, g)$, while a hypercube with $n = g^2$ processors makes $O(\log n)$ moves.

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] P. Berthomé, J. Cohen, and A. Ferreira. Embedding tori in Partitioned Optical Passive Star networks. *Fourth International colloquium on Structiral Information and Communication Complexity-Sirocco'97*, volume 1 of *Proceedings in Informatics*, Carleton Scientific, 40-52, 1997.

[3] 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.

[4] 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.

[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] V. Prasanna Kumar and V. Krishnan. Efficient template matching on SIMD arrays. *1987 International Conference on Parallel Processing*, The Pennsylvania State University Press, pp. 765-771.

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

[9] 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.

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

[11] S. Ranka and S. Sahni, *Hypercube algorithms with applications to image processing and pattern recognition.* Springer-Verlag, New York, 1990.

[12] S. Sahni, Matrix multiplication and data routing using a partitioned optical stars network. *IEEE Transactions on Parallel and Distributed Systems.*