Optical delay line memory model with efficient algorithms

John H. Reif
Duke University
Department of Computer Science
Durham, North Carolina 27708-0129
E-mail: reif@cs.duke.edu

Akhilesh Tyagi
Iowa State University
Department of Computer Science
Ames, Iowa 50011
E-mail: tyagi@cs.iastate.edu

Abstract. The extremely high data rates of optical computing technology (100 Mwords/s and upward) present unprecedented challenges in the dynamic memory design. An optical fiber loop used as a delay line is the best candidate for primary, dynamic memory at this time. However, it poses special problems in the design of algorithms due to synchronization requirements between the loop data and the processor. We develop a theoretical model, which we call the loop memory model (LLM), to capture the relevant characteristics of a loop-based memory. An important class of algorithms, ascend/descend—which includes algorithms for merging, sorting, discrete Fourier transformation (DFT), matrix transposition, and multiplication and data permutation—can be implemented without any time loss due to memory synchronization. We develop both sequential and parallel implementations of ascend/descend algorithms and some matrix computations. Some lower bounds are also demonstrated.
[S0091-3286(97)03709-4]

Subject terms: optical computing; delay line loops; algorithms.

Paper 22037 received Mar. 15, 1997; accepted for publication May 29, 1997.

1 Introduction

1.1 *Success of Very Large Scale Integration Theory*

Over the last 15 yr, very large scale integration (VLSI) has moved from being a theoretical abstraction to being a practical reality. As VLSI design tools and VLSI fabrication facilities such as the metal-oxide semiconductor system (MOSIS) became widely available, the algorithm design paradigms such as systolic algorithms,[1] which were thought to be of theoretical interest only, have been used in high-performance VLSI hardware. Along the same lines, the theoretical limitations of VLSI predicted by area-time trade-off lower bounds[2] have been found to be important limitations in practice.

1.2 *Promise of Optical Computing*

The optical computing technology is considered to be one of the several technologies that could provide a boost of two to three orders of magnitude in computing speed over the currently used semiconductor (silicon)-based technology. The field of electro-optical computing, however, is in its infancy stage; comparable to the state of VLSI technology, say, 10 yr ago. Fabrication facilities for many key electro-optical components are not widely available—instead, the crucial electro-optical devices must be specially made in the laboratories. However, a number of prototype electro-optical computing systems have been built recently, perhaps most notably at Bell Laboratories under Huang[3,4] and also at Boulder under Jordan (see later). In addition, several optical message-routing devices have been built at Boulder,[5] Stanford, and the University of Southern California[6–9] (USC). Thus optical computing technology has matured to the point that prototypical computing machines can be built in the research laboratories. But it certainly has not attained the maturity of VLSI technology. What is likely to occur in the future? The technology for electro-optical computing is likely to advance rapidly through the 1990s, just as VLSI technology advanced in the late 1970s and 1980s. Therefore, following our past experience with VLSI, it seems likely that the theoretical underpinnings for optical computing technology, namely, the discovery of efficient algorithms and of resource lower bounds, are crucial to guide its development. This seems to us to be the right moment for algorithm designers to get involved in this enterprise. A careful interaction between the architects and the algorithm designers can lead to a better-thought-out design. This paper is an attempt in that direction.

1.3 *Data Storage: Key Problem in Optical Computing*

The optical computing technology can obtain extremely high data rates beyond what can be obtained by current semiconductor technology. Therefore, to sustain these data rates, the dynamic storage must be based on new technologies that are likely to be completely or partially optical. Jordan at the Colorado Optoelectronic Computing Systems Center[10] and some other groups have proposed and used optical delay line loops for dynamic storage. In these data storage systems, an optical fiber, whose characteristics match the operating wavelength, is used to form a delay line loop. In particular, the system sends a sequence of optically encoded bits down one end of the loop and after a certain delay (which depends on the length and optical characteristics of the loop), the optically encoded bits appear at the end of the loop, to be either utilized at that time and/or once again sent down the loop. This idea of using propagation delay for data storage dates back to the use of mercury delay loops in early electronic computing systems before the advent of large primary or secondary memory storage. Jordan[10] has been able to store 10^4 bits per fiber loop with the fiber length of approximately 1 km. This was achieved in a small, low-cost prototype system with a synchronous loop without very precise temperature control. Jordan used such a delay loop system to build the second known purely optical computer (after Huang's), which can simulate a counter. Note that this does not represent the ultimate limitations on the storage capacity of optical delay loops, which could in principle provide very large storage using higher performance electro-optical transducers and multiple loops. Maguire and Prucnal[11] suggest using optical delay lines to form disk storage.

The main problem with such a dynamic storage is that it is not a random access memory. A delay line loop cannot be tapped at many points since a larger number of taps leads to excessive signal degradation. This implies that if an algorithm is not designed around this shortcoming of the dynamic storage, it might have to wait for the whole length of the loop for each data access. Systolic algorithms (Kung[1]) also exhibit such a tight interdependence between the dynamic storage and the data access pattern.

1.4 *Loop Memory Model and Our Results*

In this paper, we propose the loop memory model (LMM) as a model of sequential electro-optical computing with delay-line-based dynamic storage. The LMM contains the basic features that current delay loop systems use, as well as the features that systems in the future are likely to use. It would seem that the restrictive discipline imposed on the data access patterns by a loop memory would degrade the performance of most algorithms, because the processor might have to idle waiting for data. However, we demonstrate that an important class of algorithms, ascend/descend algorithms, can be realized in the LMM without any loss of efficiency. Note that many problems including merging, sorting, discrete Fourier transformation (DFT), matrix transposition and multiplication, and data permutation are solvable with an ascend/descend algorithm (described in Sec. 3). In fact, we give sequential algorithms covering a broad range for the number of loops required. A parallel implementation performing the optimal amount of work is also shown. The work performed by a parallel algorithm running on p processors in time T is given by $p*T$.

An ascend or descend phase takes time $O(n \log n)$ in the LMM using $\log n$ loops of sizes $1, 2, 4, \ldots, n$. Note that a straightforward emulation of a butterfly network with $O(n \log n)$ time performance requires $O(n)$ loops: n loops of size 1, $n/2$ of size 2, $n/4$ of size 4, $\ldots$, 1 of size n. It can be implemented in time $n^{1.5}$ just with two loops of sizes $\sqrt{n}$ and n each. This can be generalized into an ascend-descend scheme with time $nk+n^{1.5}2^{-k/2}$ with $2 \leq k \leq \log n$ loops. At this point in time, a loop is a precious resource in optical technology, and hence tailoring an algorithm around the number of available loops is an important capability. The k loop adaptation of the ascend/descend algorithm provides just this capability. A single loop processor takes n^2 time. A matching lower bound also exists for this case, which is derived in Sec. 6 from one tape Turing machine crossing sequence arguments. Matrix multiplication and matrix transposition can also be performed in an LMM without any loss of time.

We also consider a butterfly network with $p \log p$ LMM processors, where $1 < p \leq n$. The work (number of processors, time product) of this network for ascend-descend algorithms is shown to be $O(n \log n)$. Note that a butterfly network performs $n \log n$ work. This shows that the ascend-descend algorithms can be redesigned in such a way as not to incur any work loss due to the restrictive nature of the loop memories.

1.5 *Related Work*

Several models for virtual memory have been considered recently. The random access machine (RAM) was proposed as a simple model of sequential computation without any virtual memory in Aho et al.[12] Aggarwal et al.[13,14] introduced the first hierarchical memory model, which also incorporates block transfer. They extended[15] it to the parallel computation model PRAM. Vitter and Shriver[16] consider the PRAM case where parallel block transfers are permitted. In all these models, the algorithm optimization consists of exploiting either temporal or spatial locality given that the access to a block of size b at address x costs time $f(x)+b$, where $f(x)$ is the seek time. In our model, the synchronization of the loops (primary storage) with processor is the main objective in algorithm design. In a more fully developed optical computing system, it seems almost imperative that a memory hierarchy will exist, with either fast semiconductor memory or laser disks at the bottom of the hierarchy. An analysis of the traffic between the loop memory and secondary memory could be similar to the previous work. In the early days of computing, acoustic delay lines were used as mass storage. But using a delay line as a secondary storage medium does not pose the problems that are encountered when it is used as the primary storage. Its use as a secondary storage can be analyzed by the techniques developed for the hierarchical memory model[13] (HMM). The related work in the theory of optical computing is very sparse. Barakat and Reif[17] introduced VLSIO (electro-optical VLSI), a model of optical computing. They considered volume-time trade-offs and lower bounds in this model. We demonstrated[18] energy and energy-time product lower and upper bounds for optical computations. We introduce[19] the DFT-PRAM model, where a DFT can be computed in unit time to model the power of optical computing. We develop several $O(1)$ time algorithms in this model.

The work most related to this paper is that of Jordan,[10] Heuring et al.,[20] and Jordan.[21,22] They describe an optical delay loop system and show that a number of networks could be simulated by the use of optical delay loops. However, their work does not imply our results; in particular, they did not look at the algorithmic aspects of the loop memory systems.

Optical waveguides have been used in several optical computing systems, but primarily for computing and not storage. Daeshik et al.,[23] Jordan et al.,[24] and Kyungsook[25] use delay lines for time slot permutation and sorting. Barbarossa and Laybourn[26] use optical delay lines for processing. Wagner et al.[27] perform adaptive array processing with

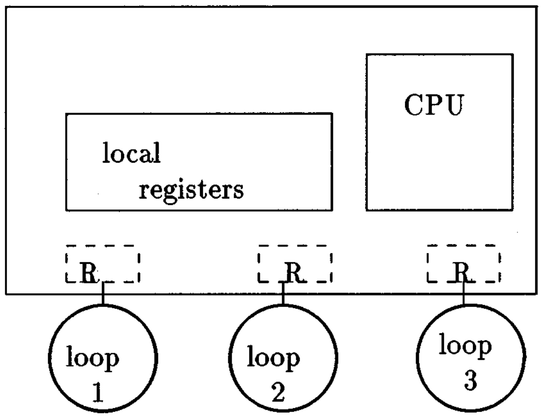


Fig. 1 Processor with three delay line memory loops.

recirculating loops. There are several applications of microwave processing with delay line loops.[28]

The algorithms presented in this paper schedule the memory loops for their data needs. However, all the problems solvable with an ascend-descend algorithm entertain structured data and hence data scheduling can be built into the algorithm as in systolic algorithms case. For random data, the memory loop scheduling (similar to register allocation in current-day compilers[29]) poses an interesting problem. Tyagi[30] proposes some algorithms for scheduling data on loops.

1.6 *Organization*

The rest of the paper is organized as follows. Section 2 introduces both the sequential processor and parallel LMM. The various sequential processor implementations of the ascend-descend algorithms are described in Sec. 3. Section 4 sketches the p processor version of ascend-descend algorithms. A brief sketch of matrix transpose and multiplication is given in Sec. 5. Some lower bounds are shown in Sec. 6. An alternative model for parallel optical computation where the delay line loops serve as both a communication link and a dynamic memory is presented in Sec. 7. Section 8 concludes the paper.

2 Model

The intent here is to develop a model for a processor with the primary dynamic storage in a delay line loop memory. The processor can have an instruction set similar to that of the RAM in Aho et al.[12] To make it a little more reasonable, we assume that the processor also contains a constant number of registers, each capable of holding one word of data. Note that a register operating at gigahertz rate (10^9/s) can be built using a very short delay line with a directional coupler.[10] Figure 1 shows a processor with three memory loops.

2.1 *Memory Loops*

A processor can have several delay line memory loops of prespecified lengths associated with it. Let there be k loops L_i of lengths n_i for $0 \leq i < k$. Note that the length of the fiber (as a multiple of the wavelength) determines the number of bits stored in it. We assume that the size of data stored in each loop L_i is a pair of words, i.e., it is a word-parallel implementation. This could mean that each conceptual loop L_i is implemented using 64 physical loops for a 32-bit word. An alternative method could be to encode 2 bits into each wave, thereby requiring only 32 physical loops. In either case, L_i can store n_i word pairs. The processor has a tap into each of these loops. A simple conceptual model is to assume that the current word under the tap is stored in a register associated with the loop. In practice, a memory loop could be directly coupled to a computational device without any need for an intermediate register. In the following, we make the assumption that for all practical purposes, the CPU accesses a loop through its tap register. A write into the loop is also performed from its tap register. We refer to the value in the tap register of the loop L_i by $v(L_i)$. We use the notation $v(L_i)=e$ to denote a write of the result of an expression e into L_i. Similarly, $x = e[v(L_i)]$ denotes a read of L_i for the evaluation of the expression e.

Note that the digital optical computer at Colorado[20,22,31] uses a bit-serial design. Each loop stores 64 words of 16-bits each in a bit-serial manner. The primary reason for choosing a bit-serial design was the cost of the optical switches. In their memory loop implementation,[31] each loop has a distinct read and write port. A lithium niobate directional coupler is used at each of these ports. Each data word recirculates serially through a delay line passing the input/output ports once each memory cycle (20.5 μs, time to go through the loop once). A memory counter keeps track of the address of the memory word at the read/write port. There are no latches in the system. The entire processor is designed with ''time-of-flight'' synchronization.

2.2 *Read/Write Delays*

The delay to read or write from a memory loop is a fraction of time required for computation. The situation is similar to the register access time being a fraction of cycle time. In this paper, we assume that both read/write require unit time. Recall that the Colorado optical computer[22] has a memory cycle of 20.5 μs with a loop storing 64, 16-bit words. If these loops were to be made word-parallel, then the time between successive words is 20 ns, which corresponds to a 50-MHz processing rate. This indeed is the processing rate of the Colorado digital optical computer. Hence our assumption about each loop access taking a machine cycle is not unrealistic. We also blur the distinction between an instruction cycle and a machine cycle in the algorithms presented in Sec. 3. We assume that instructions finish in one machine cycle as well, or that the cycles per instruction[32] (CPI) is nearly 1. The algorithms in this paper can be easily modified to fit any other delay model.

The operands for the instructions can be any of the local registers or one of the loop tap registers.

The parallel butterfly model is a direct extension of the sequential processor model. The interesting scenario occurs only when an n-input problem instance needs to be solved on a $p \log p$ processor butterfly network for $p<n$. In such a case, the loop memory at each processor is used to maintain the copies of multiple data resulting from folding. For a description of a butterfly network, see Ullman.[33]

Note that the prototype of the electro-optical computing system being built at Colorado by Jordan[10] resembles our model. All the register and main memory is realized with fiber delay line storage loops. The processing logic is also derived from the optical technology in the form of a directional coupler, where the electric field between the waveguides is used as the control mechanism.[10]

3 Ascend-Descend Algorithms

The CASCADE algorithms (ascend-descend) introduced in Preparata and Vuillemin[34] are an important class of algorithms. Consider an algorithm with $n=2^k$ input data items. Let the i'th input be located at address i for all $0 \leq i \leq (n-1)$ initially. An algorithm is an ascend algorithm if it operates successively on pairs of data items that are located $2^0, 2^1, 2^2, \ldots, 2^{k-1}=n/2$ distance apart. This involves ascending right to left on the k address bits. A descend algorithm is defined similarly to operate on pairs of data items $2^{k-1}, 2^{k-2}, \ldots, 2^0$ apart. We define CASCADE to be the class of algorithms that are composed of a sequence of ascend and descend algorithms. Many problems such as merging, sorting, DFT, matrix transposition and multiplication, and data permutation have CASCADE algorithms. Hence, an efficient implementation of ascend and descend algorithms translates into an efficient implementation of all these problems. Besides, the topology of many interconnection networks such as butterfly, perfect shuffle, hypercube, and cube-connected cycles is ideally suited for ascend/descend algorithms.

In this section, we show a single processor implementation of an ascend-descend algorithm with following resources: (1) time $O(n \log n)$, $\log n$ loops of sizes $1, 2, 4, \ldots, n$; (2) time $n^{1.5}$, two loops of sizes $\lceil\sqrt{n}\rceil$ and n; and (3) time $nk+n^{1.5}2^{-k/2}$, k loops, with $2 \leq k \leq \log n$, of sizes $(n/2^k)^{1/2}$ and $n/2^k$, $n/2^{k-1}, \ldots, n$. Note that the first implementation performs optimal amount of work $-\Theta(n \log n)$, demonstrating that an ascend/descend algorithm can be realized without any loss in time performance due to high latency evident in loop memories. Also note that the number of loops in a real optical processor might be limited. The k loop realization is helpful in this situation as it provides an implementation of ascend/descend algorithms on any optical processor.

3.1 *Loop Primitive Operations*

In the following algorithms, we perform same set of actions on loops many times. Hence we encapsulate these actions into loop primitives. We use the notation $|L_i|$ to denote the length of the loop L_i, also given by n_i. Recall that we have assumed that each location in the loop is capable of holding two words, as if there were two conceptual tracks on the loop. We refer to these words by $v_u(L_i)$ and $v_l(L_i)$ for upper and lower track words, respectively. We also refer to the words $v_u(L_i)$ and $v_l(L_i)$ by the notation $v_u(L_i^0)$ and $v_l(L_i^0)$ respectively. The notation L_i^j denotes the word that would be at the read/write head (tap register) of L_i in j steps from the current time. Figure 2 shows a loop L_i with data circulating left to right. The right end of the loop is connected to the left end. Sometimes, we need to tag a value in a loop L_i for reasoning about the correctness of our

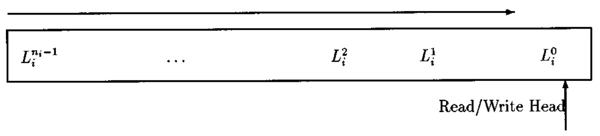


Fig. 2 Loop words.

algorithms. In that case, we assume that we can associate a memory counter with each word in the loop L_i with values between 0 and n_i-1. We refer to the upper (lower) word with memory counter value j by the notation $L_i[j]^u(L_i[j]^l)$.

Input copying. Our initial assumption is that the primary program input values reside in some other memory. We would copy different input values into a given loop (of size 1) over time. The primitive `copy_input`(x_i, L_j) copies the first argument x_i to the upper track of the loop L_j at the word aligned with the write port when this primitive is invoked, i.e., $v_u(L_j^0)=x_i$. We assume that `copy_input` takes one cycle.

Copying. In ascend-descend algorithms in the next section, many copying steps involve copying each value from a loop L_i to two locations in L_j for $|L_j|=2*|L_i|$. In particular, $v(L_i^0)$ needs to be copied to $v(L_j^0)$ and $v(L_j^{n_i-1})$. Figure 3 shows the copy pattern.

```
copy_upper($L_i$, $L_j$)
{comment: $L_j$ has twice as many words as $L_i$.}
{comment: Copy upper track words of $L_i$ into upper track of $L_j$.}

(1)$\forall k, 0 \leq k \leq n_i-1$,   $v_u(L_j^k)=v_u(L_i^k)$;

(2)$\forall k, 0 \leq k \leq n_i-1$,   $v_u(L_j^{n_i+k})=v_u(L_i^k)$;

end copy_upper
```

The second copy of L_i (through second $\forall$ at line 2) is synchronized for L_i and L_j. After line 1 has completed copying L_i into the first half of L_j, the word at $L_j^{n_i}$ at the entry into `copy_upper` is now at the read/write port (current L_j^0). At the entry into `copy_upper`, L_i^0, is again at the read/write port. We need another similar primitive for copying the upper track words from L_i to the lower track of L_j. Note that this copying takes time $n_j=|L_j|$.

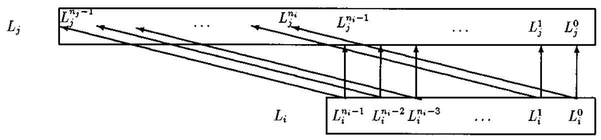


Fig. 3 Loop copying for ascend-descend algorithms.

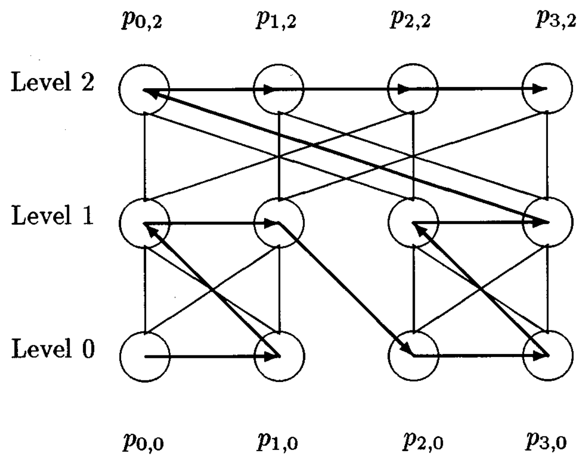


Fig. 4 Butterfly network; thick lines show the order of computing.

```
Process-Ascend($i$, $k$)
{comment: Process the $k$ input bit positions starting at $x_i$ for ascend phase. }
{comment: $k$ is assumed to be power of 2. }
    if $k = 1$ then copy_input($x_i$, $L_1$), copy input bit $x_i$ to $L_1$.                (1)
        else
          Process-Ascend($i$, $k/2$);                                                 (2)
          copy_upper($L_{k/2}$, $L_k$);                                               (3)
          Process-Ascend($i + \frac{k}{2}$, $\frac{k}{2}$);                           (4)
          copy_lower($L_{k/2}$, $L_k$);                                               (5)
          compute_loop($L_k$, $f$);                                                   (6)
           {comment: compute the values $x_{i,k} = f(x_{i,k/2}, x_{i\pm k/2,k/2})$, for $0 \le i \le (k-1)$.}
    end if
{comment: Note that $x_{i,k/2}$ and $x_{i\pm k/2,k/2}$ are available in $L_k$ at position $i$ due to previous two copying steps. }
end Process-Ascend
```

Fig. 5 Recursive version of ascend phase algorithm.

```
copy_lower($L_i$,$L_j$)
  {comment: $L_j$ has twice as many words as $L_i$.}
  {comment: Copy upper track words of $L_i$ into lower track of $L_j$.}
(1)$\forall k, 0 \leqslant k \leqslant n_i - 1, \quad v_l(L_j^k) = v_l(L_i^k)$;
(2)$\forall k, 0 \leqslant k \leqslant n_i - 1, \quad v_l(L_j^{n_i+k}) = v_l(L_i^k)$;
end copy_lower
```

Loop computation. Another step in the ascend-descend algorithms is to apply a function f of two arguments to words on the upper and lower tracks for an entire loop.

```
compute_loop($L_i$,$f$ )
  {comment: Apply $f$ to words in $L_i$.}
$\forall j, 0 \leqslant j \leqslant n_i - 1, \quad v_u(L_i^k) = f(v_u(L_i^k), v_l(L_i^k))$;
end compute_loop
```

An assumption made here is that the reads of two words $[v_u(L_i^0)$ and $v_l(L_i^0)]$, computation of f, and the write of result $v_u(L_i^0)$ all takes one cycle. If need be, the result could be shifted by a few positions determined by the latency c of the computation, i.e., $v_u(L_i^{k+c}) = f(v_u(L_i^k), v_l(L_i^k))$. The time taken by this computation is $n_i = |L_i|$, assuming one cycle read-compute-write.

3.2 *The* log *n Loop Version*

We number the levels of a butterfly network from 0 through $\log n$ (bottom-up, as in Fig. 4) such that the cross-connections between level i and $i+1$ have a span 2^i. The processors on the same level are numbered 0 through $n-1$ from left to right. Let $x_{i,j}$ for $0 \leqslant i \leqslant (n-1)$ and $1 \leqslant j \leqslant \log n$ be the value computed in the processor $p_{i,j}$. The input value x_i corresponds to $x_{i,0}$ for $0 \leqslant i \leqslant (n-1)$. Note that for the ascend phase $x_{i,j} = f(x_{i,j-1}, x_{i+2^{j-1},j-1})$ when the j'th bit from the right in the binary representation of i is 0, and $x_{i,j} = f(x_{i,j-1}, x_{i-2^{j-1},j-1})$ otherwise. Here f is some operation performed in the processors. Similarly, for the descend phase $x_{i,j}$, for $0 \leqslant i \leqslant (n-1)$ and $0 \leqslant j \leqslant (\log n - 1)$, is given by the following expressions: $x_{i,j} = g(x_{i,j+1}, x_{i+2^{j+1},j+1})$ when the $(j+1)$st bit from the right in the binary representation of i is 0, and $x_{i,j} = g(x_{i,j+1}, x_{i-2^{j+1},j+1})$ otherwise. The skeleton of the implementation is as follows. A butterfly computation graph is scanned in the order shown in Fig. 4. We demonstrate only the ascend-phase of the algorithm. The descend phase is very similar.

We assume that $\log n$ memory loops L_1, L_2, L_4, $\ldots$, L_n of sizes $1, 2, 4, 8, \ldots, n$, respectively are available. Recall that the word i positions away from the read/write port in the loop L_k is referred to as L_k^i. The zeroth position of a loop L_k^0 is the position scanned by the tap register currently. The following implementation of the ascend algorithm assigns the physical loop L_{2^i} to the storage of the values associated with butterfly level i for $0 \leqslant i \leqslant \log n$. For instance, loop L_1 is time-multiplexed to store all the butterfly level 0 values, while loop L_2 stores all the butterfly level 1 values. Figure 5 presents the ascend phase algorithm.

The ascend phase is implemented by a call to **Process-Ascend**$(0,n)$ (Fig. 5), where n is assumed to be a power of 2. It takes time $3n \log n + n$ as

$$T(n) = 2T(n/2) + 3n, \quad T(1) = 1.$$

The two recursive calls to **Process-Ascend** in steps 2 and 4 of Fig. 5 give the term $2T(n/2)$ and the two copying steps in steps 3 and 5 lead to the $2n$ term. The computation in step 6 adds another n to this equation. These equations have a solution in $T(n) = 3n \log n + n$.

The main concern in determining the correctness of this implementation and its timing analysis is the synchronization of all the loop accesses. For instance, at the return from the call **Process-Ascend**$(0,n/2)$ at step 2, are the loops $L_{n/2}$ and L_n synchronized so that copying takes place in n steps? Note that the usage pattern of the loops by this algorithm has the following form: L_1, L_2, L_1, L_2, L_4, L_1, L_2, L_1, L_2, L_4, L_8, and so on. The following inductive proof establishes that the timing of the algorithm is correct, as the-synchronization is done by ''time-of-flight.'' Note that we have not defined the term ''synchronization'' used in the statement of the following theorem

Theorem 1. The ascend phase algorithm in Fig. 5 has synchronized data transfer and computation to emulate the behavior of a butterfly network.

Proof. The proof is inductive. Assume that all the data transfers and computation are synchronized for **Process-Ascend**(i,k) for all values of $k<n$. The inductive basis for $k=1$ is trivial, as it involves only the transfer of x_i into L_1. The inductive step is to prove that **Process-Ascend**$(0,n)$ is synchronized given that **Process-Ascend**$(0,n/2)$ and **Process-Ascend**$(n/2,n/2)$ are synchronized.

Assuming that $n>1$, **Process-Ascend**$(0,n)$ first calls **Process-Ascend**$(0,n/2)$, copies $L_{n/2}$ into L_n, calls **Process-Ascend**$(n/2,n/2)$, copies $L_{n/2}$ into L_n, and finally computes L_n. After **Process-Ascend**$(0,n/2)$ is done, the assumption (part of inductive hypothesis) is that $L_{n/2}$ is scanning $x_{0,\log n-1}$, or $v_u(L^0_{n/2})=x_{0,\log n-1}$. In addition to showing that all the copying steps work correctly, we also need to prove that at the end of **Process-Ascend**$(0,n)$, $v_u(L^0_n)=x_{0,\log n}$.

Let $T=0$ when the `copy_upper` at step 3 starts copying $L_{n/2}$ into L_n. Let the memory counter of the word copied into L_n at $0\leqslant T=k\leqslant n-1$ be k. Similarly, let the memory counter in $L_{n/2}$ for the word copied at $0\leqslant T=k<n/2$ be k. Note that $L_n[j]^u=L_n[j+n/2]^u$ for $0\leqslant j<n/2$ and $L_{n/2}[0]^u=x_{0,\log n-1}$, which implies that $L_n[j]^u=L_n[j+n/2]^u=x_{j,\log n-1}$. The copying is completed at $T=n-1$, when **Process-Ascend**$(n/2,n/2)$ is called. This call is completed in $(3n/2)(\log n-1)+n/2$ steps. Once again, at time $T=n+(3n/2)(\log n-1)+n/2$, $v_u(L^0_{n/2})=x_{n/2,\log n-1}$ by inductive hypothesis. The key question from synchronization perspective is which word is at the read/write port of L_n at $T=n+(3n/2)(\log n-1)+n/2$? We would like $v_u(L^0_n)$ to be either $L_n[0]^u$ or $L_n[n/2]^u$, which holds if $T=n+(3n/2)(\log n-1)+n/2$ is a integral multiple of $n/2$. Since $n/2$ divides the time elapsed since the last access to memory counter 0 in L_n, our synchronization holds for this copying step for the following reason. After `copy_lower` of step 5 is completed at $T=(5n/2)+(3n/2)(\log n-1)$, $L_n[j]^l=L_n[j+n/2]^l=x_{n/2+j,\log n-1}$ for $0\leqslant j<n/2$ and we already had $L_n[j]^u=L_n[j+n/2]^u=x_{j,\log n-1}$. Hence when `compute_loop` of step 6 starts at time $T=(5n/2)+(3n/2)(\log n-1)$, $v_u(L^j_n)=x_{j,\log n-1}$ and $v_l(L^j_n)=x_{n/2+j,\log n-1}$ for $0\leqslant j<n/2$, as expected by the ascend algorithm, resulting in $L_n[j]^u=x_{j,\log n}$. Similarly, $v_u(L^j_n)=x_{j-n/2,\log n-1}$ and $v_l(L^j_n)=x_{j,\log n-1}$ for $n/2\leqslant j<n$, and hence $L_n[j]^u=x_{j,\log n}$. This establishes one part of the inductive hypothesis. At time $T=(7n/2)+(3n/2)(\log n-1)$, when `compute_loop` is done, $v_u(L^0_n)=x_{0,\log n}$, establishing the second part of the inductive hypothesis. □

Figure 5 gives a recursive version of the ascend phase implementation. We give an iterative version in Fig. 6 as it

```
  for ($d = 1$; $d \le n$; $d = d + 1$;)                                              (1)
{comment: $d$ is the diagonal number. }
      for ($k = 1$; $k \le d$; $k = 2 * k$;)                                          (2)
{comment: $k$ is the span of the data being processed. }
         if (($i = (d - k)$) mod $k \equiv 0$)                                        (3)
{comment: $i$ is the starting position. All valid $(k, i)$ combinations have $i \bmod k = 0$.}
            if ($k == 1$) copy_input($x_i$, $L_1$)                                    (4)
               else compute_loop($L_k$, $f$);                                         (5)
            if ($i \bmod (2 * k) \equiv 0$) copy_upper($L_k$, $L_{2*k}$)              (6)
               else copy_lower($L_k$, $L_{2*k}$);                                     (7)
```

Fig. 6 Iterative version of ascend phase algorithm.

helps understand the timing of the ascend phase from a different perspective. The key point to note here is that **Process-Ascend**(i,k) calls one computing primitive (`compute_loop` usually, or `copy_input` for an input bit with $k=1$) and one copying primitive (`copy_upper` or `copy_lower`) for each feasible value of (i,k). The temporal order in which these two primitives for different (i,k) pairs are called is as follows: (0,1),(1,1),(0,2),(2,1),(3,1),(2,2),(0,4),(4,1),(5,1),(4,2),(6,1),(7,1),(6,2),(4,4),(0,8),...,(0,n). Note that $i+k$ denotes a diagonal in the butterfly network of Fig. 4. Also note that all the (i,k) pairs are invoked in the increasing order of their diagonal $d=i+k$. Of course, some of the diagonal entries are never invoked, for instance $(i=1, k=2)$ is not used due to the nature of this computation. Line 3 in Fig. 6 checks for the validity of each (i,k) pair. Lines 1 and 2 set up these (i,k) pairs going over diagonals d from 1 to n.

The preceding description assumes that the n input words are available in some other memory (not the optical loops). This is not a realistic assumption since the processor contains only a small sized $[O(1)]$ local storage in addition to the memory loops. The only other place to hold the input is the slow secondary memory. In the following, we show that if the input words are resident in the loop L_n initially, then they can be trickled down to the smaller loops in synchrony with **Process-Ascend** without any loss of efficiency. Here, we assume that either there is a separate third track on each loop to hold the input values or there is a separate set of $\log n$ input loops. In either case, let I_k denote the input loop of size k, which is either part of the loop L_k or is an independent loop. Let the n input words be in I_n initially. In fact, we assume that the memory counter for the input word x_j is j, i.e., $I_n[j]=x_j$ for $0\leqslant j<n$. We define two more primitives for the input loops: `copy_input_left` and `copy_input_right`. The primitive `copy_input_left`$(I_k,I_{k/2},i)$ copies I_k's left half ($k/2$ words) to $I_{k/2}$ starting with input word x_i, which would be at memory counter value 0.

`copy_input_left`$(I_k,I_{k/2},i)$
{**comment:** Argument i is only for reading convenience. Whenever this primitive is called, we maintain the invariant that $I_k[0]=x_i$.}

```
        $\forall j,\ 0 \leqslant j < k/2,\ I_{k/2}[j] = I_k[j]$;
    end copy_input_left
```

We define `copy_input_right`$(I_k, I_{k/2}, i+k/2)$ to copy the second (right) half of I_k ($k/2$ words) into $I_{k/2}$. This task could have been performed by `copy_input_left`, but again for presentation clarity we also define `copy_input_right`. Once again, the specification of $i+k/2$ is really redundant. The expectation is to copy $I_k[k/2+j] = x_{i+k/2+j}$ for $0 \leqslant j < k/2$ to $I_{k/2}[j]$. The invariant maintained is that whenever this primitive is called $v(I_k^0) = I_k[k/2] = x_{i+k/2}$.

```
    copy_input_right$(I_k, I_{k/2}, i+k/2)$
    {comment: Argument $i+k/2$ is only for reading convenience. Whenever this primitive is called, $I_k[0] = x_{i+k/2}$.}
        $\forall j,\ 0 \leqslant j < k/2,\ I_{k/2}[j] = I_k[k/2+j]$;
    end copy_input_right
```

The procedure **Process-Ascend** also needs to be modified as follows:

```
Process-Ascend$(i,k)$
{comment: Process the k input bit positions starting at $x_i$ for ascend phase.}
{comment: $k$ is assumed to be power of 2.}
    if $k=1$ then return;                                  (1)
      else
        copy_input_left$(I_k, I_{k/2}, i)$;                (2)
        Process-Ascend$(i, k/2)$;                          (3)
        copy_upper$(L_{k/2}, L_k)$;                        (4)
        copy_input_right$(I_k, I_{k/2}, i+k/2)$;           (5)
        Process-Ascend$(i+k/2, k/2)$;                      (6)
        copy_lower$(L_{k/2}, L_k)$;                        (7)
        compute_loop$(L_k, f)$;                            (8)
          {comment: compute the values $x_{i,k} = f(x_{i,k/2}, x_{i\pm k/2,k/2})$, for $0 \leqslant i \leqslant (k-1)$.}
    end if
{comment: Note that $x_{i,k/2}$ and $x_{i\pm k/2,k/2}$ are available in $L_k$ at position $i$ due to previous two copying steps.}
end Process-Ascend
```

We now need to argue that this new version of **Process-Ascend** works correctly and that the input words are also synchronized. All that we have modified in **Process-Ascend** is to insert `copy_input_left`$(I_k, I_{k/2}, i)$ at line 2 and `copy_input_right`$(I_k, I_{k/2}, i+k/2)$ at line 5. Also, the action taken when $k=1$ (at line 1) has changed from copying the input word x_i into L_1 to doing nothing. This version of ascend algorithm takes time $4n \log n$. Note that

$$T(n) = 4n + 2T(n/2), \quad T(1) = 0.$$

The input copying steps at lines 2 and 5 each take time $n/2$. This results in extra n term for the expression for $T(n)$. Since nothing is done for $k=1$ case, the expression for $T(1)$ is 0. These equations have a solution in $T(n) = 4n \log n$.

We now give a proof sketch for the timing of this version. The key invariant is that right before **Process-Ascend** (i,k) is invoked I_k contains the input words $x_i, x_{i+1}, \ldots, x_{i+k-1}$ with $v(I_k^0) = x_i$. It is true for the top-level call **Process-Ascend**$(0,n)$ as an assumption. For all the other recursive calls of **Process-Ascend** this invariant is maintained by lines 2 and 5.

Lemma 1. The revised version of **Process-Ascend** is synchronized.

Proof. The proof is by induction on k as in Theorem 1. The inductive hypotheses include: (1) when **Process-Ascend**(i,k) is called, I_k contains x_{i+j} for $0 \leqslant j < k$ and $v(I_k^0) = x_i$, (2) at exit from **Process-Ascend**(i,k), $v_u(L_k^0) = x_{i,\log k}$, and (3) all the copying steps in lines 2, 4, 5, and 7 are synchronized.

Let us establish the hypothesis for **Process-Ascend**$(0,n)$ assuming it is true for all $k<n$. Note that by assumption $v(I_n^0) = x_0$ and hence copying step at line 2, copies $x_0, x_1, \ldots, x_{n/2-1}$ into $I_{n/2}$ establishing the same invariant for the **Process-Ascend**$(0,n/2)$ on line 3. Line 2 `copy_input_left` takes $n/2$ steps. Hence at the entry into **Process-Ascend**$(0,n/2)$, $v(I_n^0) = x_{n/2} = I_n[n/2]$. **Process-Ascend**$(0,n/2)$ takes time $2n(\log n - 1)$, which is an integral multiple of n. Note `copy_upper`$(L_{n/2}, L_n)$ is synchronized to the beginning of $L_{n/2}$ by hypothesis 2. This copying at line 4 takes n steps. At the end of this copying, $v(I_n^0)$ still is $x_{n/2}$ [$2n(\log n - 1) + n$ steps since entry into line 3 **Process-Ascend**$(0,n/2)$]. Hence `copy_input_left` at line 5 is entered with the correct invariant copying $x_{n/2}, \ldots, x_{n-1}$ into $I_{n/2}$, establishing hypothesis 1 for the following **Process-Ascend** $(n/2,n/2)$ call. Note `copy_lower` is synchronized by the argument used in Theorem 1 since the number of steps elapsed from the end of `copy_upper` (line 4) is $2n(\log n - 1) + n/2$, which is an integral multiple of $n/2$. Since `copy_lower` takes n steps, `compute_loop` is also synchronized leaving $v_u(L_n^0) = x_{0,\log n}$, establishing hypothesis 2. □

3.3 *Two-loop Version*

An ascend-descend algorithm can be implemented with fewer loops (two) but then it takes longer time [$O(n^{1.5})$]. There seems to be a trade-off between the number of loops deployed and the time taken by the algorithm. The early optical computers are likely to have a small number of loops and hence this trade-off provides a welcome opportunity.

The basic idea for this algorithm comes from the square implementation of a barrel shifter as described in Ullman (Ref. 33, page 69). It is shown in Fig. 7. The n bits to be shifted are stored along a $\lceil \sqrt{n} \rceil \times \lceil \sqrt{n} \rceil$ array. We number the rows bottom-up from 0 to $\lceil \sqrt{n} \rceil - 1$ and columns left-right from 0 to $\lceil \sqrt{n} \rceil - 1$. The least significant input bit x_0 is stored at (0,0) position, $x_{\lceil \sqrt{n} \rceil} - 1$ at $(\lceil \sqrt{n} \rceil - 1, 0)$, $x_{\lceil \sqrt{n} \rceil}$ at

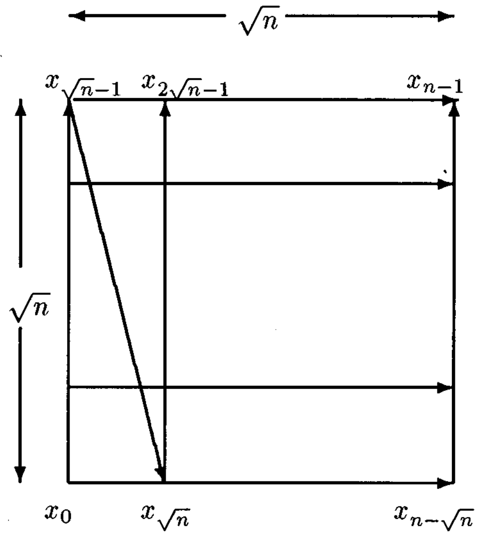


Fig. 7 Square shifter.

(0,1), and x_{n-1} at $(\lceil\sqrt{n}\rceil-1, \lceil\sqrt{n}\rceil-1)$. In the following, we use $\sqrt{n}$ instead of $\lceil\sqrt{n}\rceil$ to reduce the clutter of notation. The $\log n$ control bits $c_{\log n-1}c_{\log n-2}...c_1c_0$ form the word specifying the shift amount $0\leq s<n$. The least significant $(\log n)/2$ control bits specify the vertical shift amount. This is because the bits $c_{(\log n/2)-1}\cdots c_1c_0$ encode a shift amount $0\leq s_v<\sqrt{n}$. Note that any bits shifted out of column j for $0\leq j<\sqrt{n}-1$ move into appropriate entry of column $(j+1)$. Similarly, the most-significant half control bits $c_{\log n-1}...c_{(\log n)/2}$ specify the horizontal shift amount. Note that each 1-bit shift horizontally corresponds to a shift by $2^{\log n/2}=\sqrt{n}$ in the original word. This shifting takes time $O(\sqrt{n})$, in particular, at most $2\sqrt{n}$.

For an ascend-descend algorithm, the processing at level j (as shown in Fig. 4) requires communication between words $x_{i,j-1}$ and $x_{i\pm 2^{j-1},j-1}$. This communication is achieved through a square-shifter emulating mechanism in the two-loop version. We use one loop of size n to store all the n words, entire array of the square shifter. This loop is called main loop. Another loop of size $\sqrt{n}$ is used to process one column or row of the square shifter array at a time. We refer to the $\sqrt{n}$ size loop by context loop. The input data is initially stored in the main loop L_n. Note that any row or column can be copied from the main loop L_n to the context loop $L_{\sqrt{n}}$ within n steps.

We process the $\sqrt{n}$ columns first followed by the $\sqrt{n}$ rows. We copy each column and row from the main loop into the context loop in turn and process it. The processing of column i for $0\leq i<\sqrt{n}$ corresponds to processing lower $\log n/2$ levels (levels 1—$\log n/2$) for the columns $i*\sqrt{n}$—$(i+1)*\sqrt{n}-1$ of the butterfly network. Similarly, the computation of the j'th row in the square shifter (for $0\leq j<\sqrt{n}$) processes upper $\log n/2$ levels (levels $\log n/2+1$—$\log n$) for the columns j, $j+\sqrt{n}$, ..., $n-\sqrt{n}+j$. Note that if n is chosen to be a power of 2, all the words needed by level i computation (for $1\leq i<\log n/2$) are contained in loop $L_{\sqrt{n}}$. If n were not a power of 2, the context loop would need to be $2\sqrt{n}$ long to be able to contain entries for columns i and $i+1$ when column i is being processed. For instance, for $n=9$, the context loop has three entries. When level 1 (see Fig. 4) is computed, $x_{0,0}$ and $x_{1,0}$ should communicate (columns 0 and 1 of butterfly) and $x_{2,0}$ and $x_{3,0}$ should communicate (columns 2 and 3 of the butterfly). However, the context loop contains only the values of $x_{0,0}$, $x_{1,0}$,$x_{2,0}$. If we did use the context loop of size $2\sqrt{n}$, which is 6 in this case, the context loop would have values $x_{0,0}$, $x_{1,0}$,...$x_{5,0}$. When column 0 of the square shifter is processed, the values for $x_{0,1}$, $x_{1,1}$, $x_{2,1}$, and $x_{3,1}$ will be computed. The values for $x_{4,1}$ and $x_{5,1}$ are computed when column 1 of square shifter is handled. In general, in this case $2^{\lceil\log n\rceil/2}>\sqrt{n}$. We compute $k=2^{\lceil\log n\rceil/2}$ values for each iteration of the context loop for a square shifter column (except the last one). For simplicity of illustration, we assume in the following that n is a power of 2.

For the two-loop algorithm, we are assuming the synchronization of the loops is done through memory counters for the loops. Let us assume that the i'th butterfly column value $x_{i,l}$ for all levels l is at memory counter i for L_n. We use the upper track on $L_{\sqrt{n}}$ to keep the level $l+1$ values computed from the level l values in the lower tracks. For the square shifter column i, resulting in butterfly columns $i\sqrt{n}$, $i\sqrt{n}+1$, ...,$(i+1)\sqrt{n}-1$ being copied into $L_{\sqrt{n}}$, the memory counter values are 0, 1, ..., $\sqrt{n}-1$, respectively. The primitives are

```
copy_column($i$,$L_n$,$L_{\sqrt{n}}$)
{comment: copies square shifter column $i$ from $L_n$ to $L_{\sqrt{n}}$. Either track can be used on $L_n$ and the lower track is used on $L_{\sqrt{n}}$.}

$\forall j$, $0\leq j<\sqrt{n}$, $L_{\sqrt{n}}[j]^l=L_n[i\sqrt{n}+j]$;

end copy_column
```

The primitive `copy_column` can take up to $n+\sqrt{n}-1$ steps. Thus, L_n may have to wait for up to $n-1$ steps for the L_n memory counter to get to $i\sqrt{n}$ and $\sqrt{n}$ steps are needed to copy after that. We also need a primitive to move the computed values back from the upper track of $L_{\sqrt{n}}$ to L_n.

```
move_column_to_main($i$,$L_n$,$L_{\sqrt{n}}$)
{comment: copies upper track of $L_{\sqrt{n}}$ to column $i$ of $L_n$.}

$\forall j$, $0\leq j<\sqrt{n}$, $L_n[i\sqrt{n}+j]=L_{\sqrt{n}}[j]^u$;

end move_column_to_main
```

`move_column_to_main` also takes up to $n+\sqrt{n}-1$ steps. The following primitive `copy_row`(i,L_n,$L_{\sqrt{n}}$) copies i'th row from L_n to $L_{\sqrt{n}}$.

```
copy_row($i$,$L_n$,$L_{\sqrt{n}}$)
```

```
for $i = 0$ up to $\sqrt{n} - 1$ do                                                        (1)
{comment: Handle the ith column.}
  copy_column($i$, $L_n$, $L_{\sqrt{n}}$);                                                  (2)
  for $l = 1$ up to $\frac{\log n}{2}$ do                                                   (3)
{comment: Each level (lth) of the butterfly network is processed in turn. }
{comment: First deal with the lower $\frac{\log n}{2}$ levels corresponding to column operations. }
      for $j = 0$ up to $\sqrt{n} - 1$ do                                                   (4)
{comment: The value at jth position interacts with the value at $j \pm 2^{l-1}$. }
          compute_value($x_{i\sqrt{n}+j,l}$, $2^{l-1}$, $L_{\sqrt{n}}$, $j$, $f$);          (5)
{comment: this is done in one rotation of $L_{\sqrt{n}}$.}
      end for
    end for
  move_column_to_main($i$, $L_n$, $L_{\sqrt{n}}$);                                          (6)
{comment: copy the updated values back to the main loop before getting next column.}
end for
{comment: Now deal with the upper $\frac{\log n}{2}$ levels corresponding to row operations. }
for $i = 0$ up to $\sqrt{n} - 1$ do                                                         (7)
  copy_row($i$, $L_n$, $L_{\sqrt{n}}$);                                                     (8)
  for $l = (\log n)/2 + 1$ up to $\log n$ do                                                (9)
{comment: process upper $\log n/2$ levels.}
      for $j = 0$ up to $\sqrt{n} - 1$ do                                                   (10)
{comment: The value at jth position interacts with the value at $j \pm 2^{l-\frac{\log n}{2}}$. }
          compute_value($x_{j\sqrt{n}+i,l}$, $2^{l-1-((\log n)/2)}$, $L_{\sqrt{n}}$, $j$, $f$);  (11)
{comment: this is done in one rotation of $L_{\sqrt{n}}$.}
      end for
    end for
  move_row_to_main($i$, $L_n$, $L_{\sqrt{n}}$);                                             (12)
end for
```

Fig. 8 Two-loop version of ascend phase algorithm.

{**comment:** copies square shifter row i from L_n to $L_{\sqrt{n}}$. Either track can be used on L_n and the lower track is used on $L_{\sqrt{n}}$.}

```
for ($j=0;j<\sqrt{n};j=j+1$)
    $L_{\sqrt{n}}[j]^l = L_n[i+j\sqrt{n}]$;
  end copy_row
```

Note that `copy_row` can take up to $2n-1$ steps, $n-1$ for the L_n memory counter synchronization and n steps for the copying. Again, we need a reverse primitive to move the upper track of a row from $L_{\sqrt{n}}$ back to L_n.

`move_row_to_main`(i,L_n,$L_{\sqrt{n}}$)
{**comment:** copies the computed values in row i from the upper track of $L_{\sqrt{n}}$ to L_n.}

```
for ($j=0;j<\sqrt{n};j=j+1$)
$L_n[i+j\sqrt{n}] = L_{\sqrt{n}}[j]^u$;
  end move_row_to_main
```

`move_row_to_main` can also take up to $2n-1$ steps. The following primitive computes a level l value from two level $l-1$ values in $L_{\sqrt{n}}$.

`compute_value`($x_{j,l}$, k, $L_{\sqrt{n}}$, j', f)
{**comment:** compute $f(x_{j,l-1}, x_{j\pm k,l-1})$ and put it into $L_{\sqrt{n}}[j']^u$ as $x_{j,l}$ for $l=\log k$.}

```
if ($j$ mod $2*k \equiv 0$)
    $L_{\sqrt{n}}[j']^u = f(L_{\sqrt{n}}[j']^l,\ L_{\sqrt{n}}[j'+k]^l)$;
else $L_{\sqrt{n}}[j']^u = f(L_{\sqrt{n}}[j']^l,\ L_{\sqrt{n}}[j'-k]^l)$;
    end compute_value
```

This version of `compute_value` takes up to $\sqrt{n}$ steps collecting the two level $l-1$ values. Figure 8 shows the algorithm for the two-loop version.

This algorithm takes time $n^{1.5}\log n + 6n^{1.5}$. The $6n^{1.5}$ term is due to copying of context to and from the main loop. This is because for each level of butterfly, each row/column takes time proportional to n. It can be easily converted into a $O(n^{1.5})$ algorithm as follows. In `compute_value`, waiting for one rotation to access x_{i+k} is wasteful. If we pipeline this computation, then in one rotation of the context loop $L_{\sqrt{n}}$,($\sqrt{n}/2k$) rather than one value can be computed. For instance, for the l'th level with column 0, in the first rotation $x_{0,l}$, $x_{2^l,l}$, $x_{2^{l+1},l}$, . . . , $x_{\sqrt{n},l}$. Thus for column (row) computation, the total computation time per column (row) is $\Sigma_{i=1}^{\log n/2} 2^i \sqrt{n}$, which is n. This gives a total time of $2n^{1.5} + 6n^{1.5}$. The term $2n^{1.5}$ in the pipelined case compares favorably with the term $n^{1.5}\log n$.

To implement the pipelined version, the algorithm in Fig. 8 would have to be modified at two places. Lines 4 and 5 would be replaced by

```
for $j=0$ up to $2^{l-1}$ do                                    (4)
compute_pipelined_value
  ($x_{i\sqrt{n}+j,l}$, $2^{l-1}$, $L_{\sqrt{n}}$, $j$,$f$ );   (5)
end for
```

Similarly, lines 10 and 11 are replaced by

```
for $j=0$ up to $\sqrt{n}-1$ do                                 (10)
compute_pipelined_value
  ($x_{j\sqrt{n}+i,l}$, $2^{l-1-((\log n)/2)}$, $L_{\sqrt{n}}$, $j$, $f$ );   (11)
end for
```

The new primitive `compute_pipelined_value` ($x_{j,l}$, k, $L_{\sqrt{n}}$, j', f) computes $f(x_{j,l-1}, x_{j\pm k,l-1})$, $f(x_{j+k,l-1}, x_{j+k\pm k,l-1})$, $f(x_{j+2k,l-1}, x_{j+2k\pm k,l-1})$,... in one rotation of the loop $L_{\sqrt{n}}$. It is is defined as follows:

```
  compute_pipelined_value
($x_{j,l}$, $k$, $L_{\sqrt{n}}$, $j'$, $f$)
for($start=j$, $mc=j'$; $mc$
        $<\sqrt{n}; mc=mc+k$, $start=start+k$;)
if ($start$ mod $2*k \equiv 0$)
$L_{\sqrt{n}}[mc]^u = f(L_{\sqrt{n}}[mc]^l,\ L_{\sqrt{n}}[mc+k]^l)$;
```

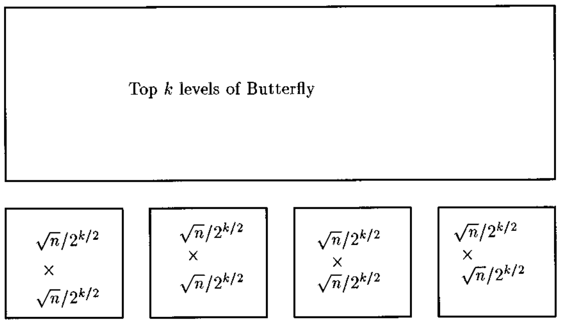

Top k levels of Butterfly
$\sqrt{n}/2^{k/2} \times \sqrt{n}/2^{k/2}$
$\sqrt{n}/2^{k/2} \times \sqrt{n}/2^{k/2}$
$\sqrt{n}/2^{k/2} \times \sqrt{n}/2^{k/2}$
$\sqrt{n}/2^{k/2} \times \sqrt{n}/2^{k/2}$

Fig. 9 A k loop algorithm.

```
        else $L_{\sqrt{n}}[mc]^u = f(L_{\sqrt{n}}[mc]^l,\ L_{\sqrt{n}}[mc-k]^l)$;
    end for
end compute_pipelined_value
```

Note that this algorithm also provides a $O(n^2)$ one loop implementation. The processor contains only one loop of length n. The i'th level of butterfly takes time $2^i n$ for $1 \leq i \leq \log n$.

3.4 *A k Loop Version*

What if we only had $2 \leq k+1 < \log n$ loops available? A hybrid algorithm based on the preceding two algorithms works in time $nk + n^{1.5}2^{-k/2}$. The k loops of sizes $n/2^k$, $n/2^{k-1}$, $n/2^{k-2}$, $\ldots$, n and one loop of size $\sqrt{n}/2^{k/2}$ are required. Figure 9 demonstrates the scheme behind the algorithm. The bottom $\log n - k$ levels of the butterfly network are processed in the two-loop, mesh-based algorithm with the loops $L_{n/2^k}$ as the main loop and $L_{\sqrt{n}/2^{k/2}}$ as the context loop. Each mesh is $\sqrt{n}/2^{k/2} \times \sqrt{n}/2^{k/2}$ requiring the main loop of size $n/2^k$ and a context loop of size $\sqrt{n}/2^{k/2}$. The two-loop algorithm for each mesh takes time $8n^{1.5}/2^{1.5k}$. Hence total time taken in the two-loop phase for all the $n/2^k$ meshes is $(8n^{1.5}/2^{k/2})$. The top k levels of the butterfly are processed using the diagonal scan algorithm mentioned earlier. This phase takes $4nk$ steps. Thus the total time is bounded by $4nk + 8n^{1.5}2^{-k/2}$ steps.

4 Parallel Ascend-Descend Algorithm

All the algorithms presented so far have assumed that a single processor is used to solve the problem. In this section, we are interested in using the optical fiber loops for local storage for processors that are part of a parallel computer. In particular, we assume that we have a p-processor machine organized with butterfly connectivity. The problem occurs when an ascend-descend problem with n input words such that $n > p$ is solved on this butterfly network of p nodes. In such a case, multiple instances of data need to be stored at each processor. One loop of size n/p per processor seems to suffice to hold this data except for the bottom level processors, which require $\log n - \log p$ loops.

Recall that a p-node butterfly has $\log p + 1$ rows and each row has p processors. As stated earlier, we number the

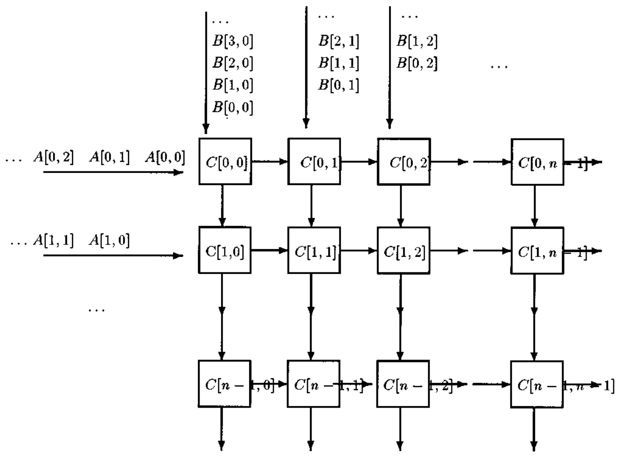


Fig. 10 Systolic algorithm for matrix multiplication.

i'th processor in the j'th row $p_{i,j}$ for $0 \leq i < p$ and $0 \leq j \leq \log p$. Note that we have a $\log p + 1$ level, p-node butterfly, for $1 \leq p < n$. For the time being, let us assume that n is a multiple of p. Let $p'_{i,j}$ refer to the i,j'th processor in the n-node computation graph for $0 \leq i < n$ and $0 \leq j \leq \log n$. The following mapping from $p'_{i,j}$ to $p_{k,l}$ will be used in our algorithm. This mapping has been used in earlier work on parallel algorithms. Note $p'_{i,j}$ for $j \leq \log(n/p)$ (lower $\log n - \log p$ levels) maps into $p_{\lfloor i/(n/p) \rfloor,0}$. For $j > \log(n/p)$, $p'_{i,j}$ maps into $p_{\lfloor i/(n/p) \rfloor, j - \log n + \log p}$. The bottom $\log n - \log p$ levels of the n-word ascend-descend problem are solved sequentially at one of the level 0 processors. Levels $l + \log n - \log p$ of the ascend-descend problem are mapped into level l of the butterfly network.

Once again, let us consider the implementation of the ascend phase. Each processor in the bottom level of the p-node butterfly simulates an n/p-node butterfly sequentially, as shown in Sec. 3, Algorithm 1. Thus, each of these p processors contains $\log n - \log p$ loops of sizes $1, 2, 3, \ldots, n/p$. The time taken for this processing is $4(n/p)\log(n/p)$. The top $\log p$ levels of the n-node butterfly are pipelined in the p-node butterfly. The j'th level of the p-node butterfly processes the $\log(n/p) + j$'th level of the n-node butterfly. Each processor $p_{k,l}$ contains n/p consecutive data values as dictated by the mapping from $p'_{i,j}$ to $p_{k,l}$. The connectivity of the p-node butterfly is adequate to emulate the connections of the n-node butterfly. To see this, note that the span of the connections between level j and $j+1$ in an n-node butterfly is 2^j. The j'th level is mapped into the $i = j - \log(n/p)$'th level of the p-node butterfly. The span of connections between the i'th and $i+1$st levels is $2^i/(n/p)$. Thus the l'th element in $p_{j,i}$ [which would have been in $p'_{(jn/p)+l, i+\log(n/p)}$] has a cross-connection with the l'th element of $p_{j \pm 2^i, i+1}$ [which would be $p'_{n/p[j \pm 2^i]+l, i+1+\log(n/p)}$.] This is the right cross-connection. The loops between two consecutive levels are also synchronized. This can be proven formally with an

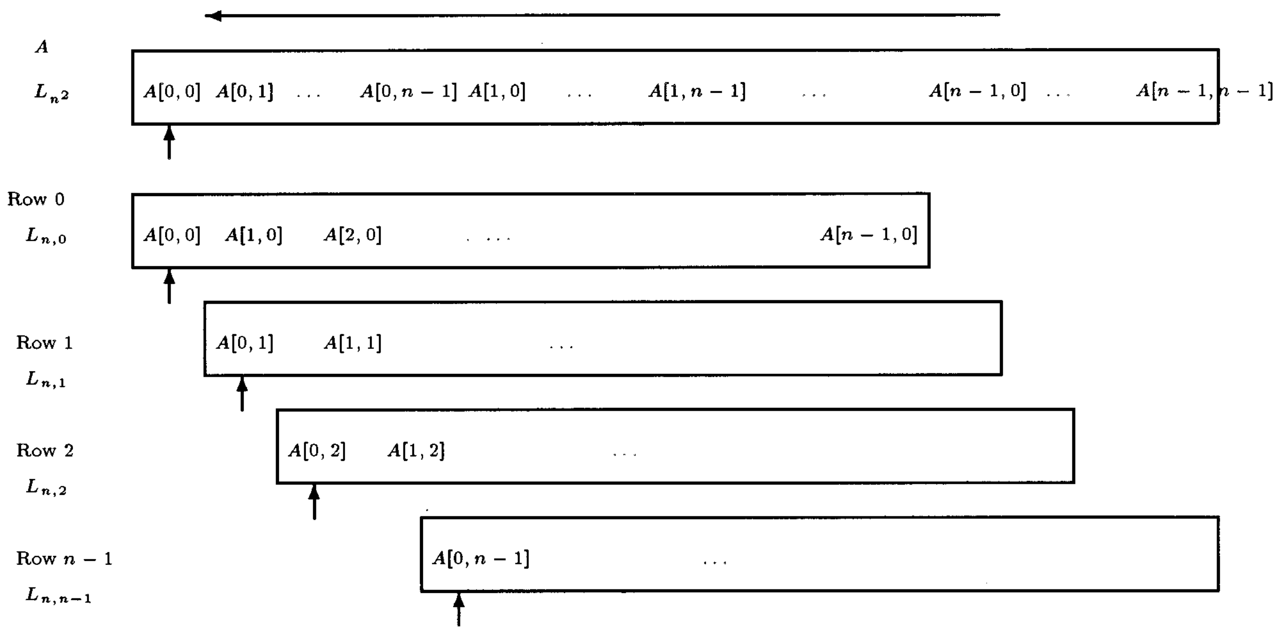

A
L_{n^2}
A[0, 0] A[0, 1] ... A[0, n − 1] A[1, 0] ... A[1, n − 1] ... A[n − 1, 0] ... A[n − 1, n − 1]
Row 0
$L_{n,0}$
A[0, 0] A[1, 0] A[2, 0] ... A[n − 1, 0]
Row 1
$L_{n,1}$
A[0, 1] A[1, 1] ...
Row 2
$L_{n,2}$
A[0, 2] A[1, 2] ...
Row n − 1
$L_{n,n-1}$
A[0, n − 1] ...

Fig. 11 Flawed scheme for transposition.

induction argument. The top $\log p$ levels take time $\log p + n/p$.

The total time taken by this algorithm is $4(n/p)\log(n/p)+\log p+n/p$. Then the work of this algorithm is $O(n \log n)$, which is the work performed by an n-node butterfly network as well.

5 Matrix Computations

The matrix multiplication of two $n \times n$ matrices can be accomplished in $O(n^3)$ steps, while the transpose takes $O(n^2)$ time. Most matrix computations are very similar to their systolic counterparts.[1] In a systolic algorithm, there are several active data streams interacting with each other. Each element of a data stream could be interacting with a distinct data stream in parallel. For instance, a commonly used systolic algorithm for matrix multiplication uses the data streams as shown in Fig. 10. Two $n \times n$ matrices **A** and **B** are multiplied in time $O(n^3)$ by this algorithm. Processor $p_{i,j}$ accumulates the value for $\mathbf{C}[i,j]$ such that $\mathbf{C}=\mathbf{A}\times\mathbf{B}$. The i'th row of **A** is input from the left edge of the array (Fig. 10) such that $(i+1)$'st row is one time step behind i'th row. For instance, $A[0,0]$ arrives at $p_{0,0}$ at $T=0$, but $A[1,0]$ arrives at $p_{1,0}$ at $T=1$. Similarly, j'th column of **B** arrives at $p_{0,j}$ at time $T=j$. The rows of **A** and columns of **B** keep marching down horizontally and vertically respectively, while $p_{i,j}$ collects the product term $A[i,k]*B[k,j]$ one by one. This systolic algorithm could be emulated in loop-based processors by allocating a loop to each data stream ($2n$ data streams in this case, one for each row of **A** and one for each column of **B**). However, the values for $C[i,j]$ are static—sitting in one place. Where should the matrix **C** be mapped? In general, all the systolic algorithms consisting of only moving data streams can be mapped to a loop algorithm. Most of the time, a static data stream (such as matrix **C**) in this example, can also be mapped on a loop with proper synchronization. Several transformations for systolic algorithms exist to convert spatial data streams (such as **C**) into temporal (moving) data streams or vice versa.[35–38]

There is one problem that seems to occur in synchronization of data streams for several matrix operations. In the following, we present the problem along with a solution. We use matrix transposition as the example for the illustration of this problem. It appears as if for transposition, one loop of size n^2 and n loops of size n each would give an obvious n^2 time algorithm. The outline of the algorithm could be as follows. Initially, the $n \times n$ matrix **A** is stored in the row-major order in the loop L_{n^2}. Each n-size loop accumulates one row of the transposed matrix $\mathbf{A}^T$ (which is a column of **A**) during the rotation of the main loop, L_{n^2}. On the surface, this step seems to be trivial. Let us illustrate it with Fig. 11. The matrix **A** is stored in row-major order on the loop L_{n^2} initially. Let us assume that $L_{n^2}[0]=A[0,0]$, with the data recirculating right to left. There are n loops of sizes n each for accumulating the i'th row of $\mathbf{A}^T$ (in $L_{n,i}$ in Fig. 11). Let us evaluate the synchronization requirements of this problem. Assume at $T=0$, L_{n^2} scans $A[0,0]$, $L_{n^2}[0]=A[0,0]$, which can be copied to $L_{n,0}[0]$ at $T=0$. Similarly, at $T=k$ for $0 \leq k < n$, we can place $A[0,k]$ from $L_{n^2}[k]$ into $L_{n,k}[0]$. The problem occurs at the copy of the next row of **A** starting at time $T=n$. At $T=n$, L_{n^2} is scanning $A[1,0]$, which goes to $L_{n,0}[1]$. However, at $T=n$, $L_{n,0}$ is scanning $L_{n,0}[0]$. The same problem occurs at $T=n+j$ for $0 \leq j < n$ since $L_{n,j}$ is scanning $L_{n,j}[0]$, while we need to access $L_{n,j}[1]$. In short, L_{n^2} and $L_{n,j}$ are not synchronized. A solution for it may be to read from L_{n^2} at

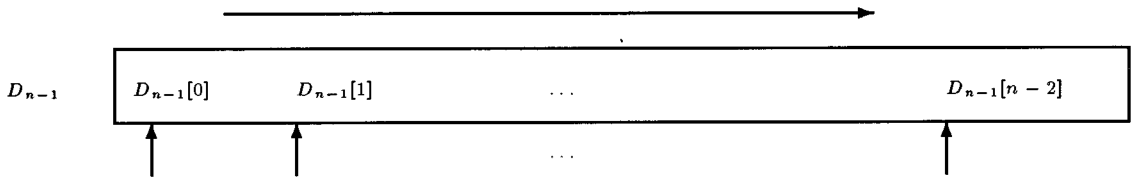


Fig. 12 Delay loop for time-shifting words.

time $T=j$ and write it into $L_{n,j}[1]$ at time $T=n+j+1$. However, the problem gets worse for the third entry or at time $T=2n+j$ for $0\leqslant j<n$. We scan $A[2,j]$ in L_{n^2}, which goes to $L_{n,j}[2]$, but we scan $L_{n,j}[0]$ at this time. In general, at time $T=i*n+j$, we need to scan $L_{n,j}[i]$, but we scan $L_{n,j}[0]$. In other words, L_{n^2} runs ahead of $L_{n,j}$ by i words for $T=i*n+j$ for $0\leqslant i,j<n$. One solution for this problem is to introduce $n-1$ delay loops of sizes 1, 2, 3, . . . ,$n-1$. The other solution is to have just one delay loop of size $n-1$ with $n-1$ taps into it. Each of these taps requires a directional coupler switch. However, a delay loop seems to be very useful for synchronization problems encountered in matrix computations. We assume that we have a single delay loop D_{n-1} with $n-1$ taps. The word at tap i is referred to by $D_{n-1}[i]$ for $0\leqslant i<n-1$. The word $D_{n-1}[0]$ is the word written at current time, $D_{n-1}[1]$ is the word written one time unit earlier and so on. This loop acts as a shifting loop. Figure 12 shows this loop. Now the transposition can be done as follows:

```
for($i=0$; $i<n$; $i=i+1$)
   for($j=0$; $j<n$; $j=j+1$)
      if ($i==0$) copy $L_{n^2}[j]$ to $L_{n,j}[0]$;
       else
          copy $L_{n^2}[i*n+j]$ to $D_{n-1}[0]$.
          copy $D_{n-1}[i]$ to $L_{n,j}[j]$.
      end if
   end for
end for
```

This version of transposition takes time n^2 to get all the rows of $\mathbf{A}^T$ into n loops of size n. If these rows were to be copied back into an n^2 size loop, this would take another n^2 steps. An alternative is to use a butterfly network with 2 log n loops. As shown in Stone,[39] for a matrix stored in row-major order, the transpose corresponds to log n shuffle steps. This can be accomplished in log n ascend-descend steps. The sequential version of ascend-descend then can transpose a matrix in $4n \log^2 n$ steps.

Matrix multiplication can similarly be performed in n^3 steps. For the computation of $\mathbf{C}=\mathbf{A}\times\mathbf{B}$ where $n\times n$ matrices $\mathbf{A}$ and $\mathbf{B}$ are initially stored in row-major order on loops $L_{n^2,\mathbf{A}}$ and $L_{n^2,\mathbf{B}}$, respectively, the following straightforward method can be used. Let us assume that $\mathbf{C}$ is to be stored on $L_{n^2,\mathbf{C}}$ in row-major order as well. First transpose $\mathbf{B}$ into n, n-size loops $L_{n,0}, L_{n,1}, \ldots, L_{n,n-1}$ as described earlier. The values of $\mathbf{C}$ are computed in the order of columns with column 0 first. We need to use the delay loop D_{n-1} again to delay the writes of columns j by j time steps for $1\leqslant j\leqslant n-1$. The following steps describe this multiplication algorithm:

```
Transpose B so that the $i$'th column of B is in $L_{n,i}$ for $0 \leqslant i<n$.
for($j=0$; $j<n$; $j=j+1$)
   for($i=0$; $i<n$; $i=i+1$)
      $X=0$;
      for($k=0$; $k<n$; $k=k+1$)
          $X=X+L_{n^2,A}[i*n+k]*L_{n,j}[k]$;
      end for
      if ($j==0$)$L_{n^2,C}[n*i]=X$;
       else
          $D_{n-1}[0]=X$; $L_{n^2}[i*n+j]=D_{n-1}[j]$;
      end if
   end for
end for
```

This matrix multiplication takes time $O(n^3)$ with n, n-size and 3, n^2-size loops. Most of the systolic matrix algorithms in Mead and Conway (Ref. 40, pages 271–285) map into loop algorithms very nicely.

6 Lower Bounds

This model offers a rich milieu for lower bounds. As we saw, a higher number of loops seems to lead to a lower time. Not just the number of loops, but also the period of data circulation in the loops is important. Ideally, we need a combinatorial technique that can decompose the data access pattern of a problem into some canonical periods. Unfortunately, at this time, we do not have such a technique. However, we do have some simple lower bounds that are tight for their subcases. Let us first consider the single-loop version.

Note that a one-loop processor is like a single-tape Turing machine, where the head is forced to move in the same direction at each time step. The tape is also wrapped around to form a simple loop. The crossing sequence arguments given in Hennie[41,42] can be used to show that the ascend-descend algorithms need $\Omega(n^2)$ time on a one-loop processor. The number of distinct input patterns is 2^n, assuming a binary input. Note that $\Omega(n)$ crossing sequences can be shown to have length $\Omega(n)$, leading to the lower bound.

The second technique quantifies the lower bound in terms of the size of the smallest loop. The assumptions are that there is only a constant amount of ''register'' storage available to the processor in addition to the loop memories. Note that all these loops have exactly one read/write port, unlike the delay loop D_{n-1} for matrix operations. The following lower bounds hold only if there is exactly one observation point into the loop memory (no loops of D_{n-1} type are available). We can derive a tight lower bound for the k loop version with this method. Let the smallest loop be of size s. The computation of lower $\log s$ levels of the ascend-descend phases can be shown to require $\Omega(sn)$ time. For the k-loop version, $s=\sqrt{n}2^{-k/2}$. This gives a lower bound of $\Omega(n^{1.5}2^{-k/2})$, which matches the time of the algorithm. Similarly, for the two-loop version, the smallest loop has size $\sqrt{n}$ and hence the lower bound translates into $\Omega(n^{1.5})$. This is also a tight bound.

Theorem 2. Let an optical computing system consisting of $k>1$ loops have the smallest loop of size s. The computation of an n-word ascend-descend algorithm requires time $\Omega(sn+n \log n)$.

Proof. We assume that the data words at level l are stored in a memory loop in the butterfly column order $x_{0,l}, x_{1,l}, \ldots, x_{n-1,l}$. For the lower bound purposes, any order that is a permutation of $(0, 1, \ldots, n-1)$ will do, but the proof is more understandable with this order. Another claim is that the lower $m=\lfloor \log s \rfloor$ levels of the butterfly network are best computed in the smallest sized (s) loop. The following discussion will prove this point.

Level l computation for ascend phase consists of $x_{i,l} = f(x_{i,l-1}, x_{i\pm 2^{l-1},l-1})$. There are $n/2^l$ groups of words consisting of 2^l words each that define the period of access required by the computation. A group of words consists of $x_{i*2^l,l}$ for $0\leq i<n/2^l$. In this group, the words $x_{i*2^l+j,l}$ for $0\leq j<2^{l-1}$ require the values of $x_{i*2^l+j,l-1}$ and $x_{i*2^l+2^{l-1}+j,l-1}$. We call this communication backward communication (with respect to the direction of data circulation in the loops). Similarly, the words $x_{i*2^l+2^{l-1}+j,l}$ for $0\leq j<2^{l-1}$ require the values of $x_{i*2^l+j,l-1}$ and $x_{i*2^l+2^{l-1}+j,l-1}$. This is the forward communication. Both forward and backward communication pose different set of problems. For instance, they have different periods. For forward communication, $x_{i*2^l+j,l-1}$ is needed at $x_{i*2^l+2^{l-1}+j,l}$ with a period of 2^{l-1} and a size of 2^{l-1}. Backward communication on the other hand requires $x_{i*2^l+j,l-1}$ to be available next time we scan $x_{i*2^l+j,l}$. The next scan of $x_{i*2^l+j,l}$ happens in $n-2^{l-1}$ time steps, which is the period of backward communication. Exactly $n-2^{l-1}$ such values must be remembered for the next scan of the level l data and hence that is also the size of the backward communication.

If we compute the lower $m=\log s$ levels of butterfly (for $1\leq l\leq m$) on the smallest loop L_s, the forward communication requirement limits us as follows. We need loops of sizes 1, 2, 4, and 2^{m-1} so that forward communication does not require additional scans of the level $l-1$ and level l data in L_s. However, $s=2^m$ is the smallest size loop available. What is the penalty of this limitation? Since only a constant amount of register storage is available on the processor, for each level l we can remember at most a constant number of words to carry forward. Let this constant be 1

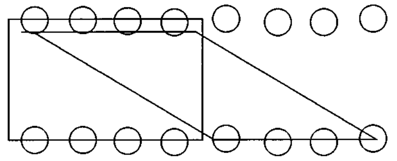


Fig. 13 Use of memory loops as communication links.

without loss of generality. Hence, we can compute at most $n/2^l$ level l values in each scan of L_s. In fact, this is exactly what is done by the two-loop, pipelined computation described earlier. Hence we need at least $\Sigma_{l=1}^{\log s}2^l$ scans of L_s to compute s of the n lower $\log s$ level values. This gives at least $2s$ scans of L_s for the computation of s values. Hence, the computation of n words for lower $\log s$ levels requires $n/s*2s=2n$ scans of L_s. Since each scan of L_s takes s time steps, the time needed for $2n$ scans is at least $2sn$. Is this the lowest time for computing lower $\log s$ levels of the ascend phase? The answer is that any other loop (of larger size) would take at least $2sn$ steps as well. Let us consider computing these lower $\log s$ levels in loop $L_{s'}$ for $s'>s$. The number of scans for computing s values from the lower $\log s$ level would again be at least $2s$. The number of iterations for computing all the n words would be n/s and hence the total time would be at least $2ns'$, which is no lower than $2ns$.

Note that we did not even consider the backward communication requirements to get this lower bound. We can derive the $n \log n$ lower bound based on the structure of the butterfly network. Since we are dealing with a sequential computation model, and there are $n \log n$ computation nodes in the butterfly graph for an ascend phase, the lower bound follows trivially.

7 Alternative Parallel Loop Model

In Sec. 4, we used the loop memory for holding the data blocks due to problem folding. The communication links between processors in a butterfly network are also implemented in fiber technology to keep up with the data rates. We propose another way of using the fiber loops so as to serve as both the data holders and the communication links. The disadvantage of this approach is that it requires many taps into a fiber. Consequently, at this point in time, this approach is not technically feasible. Figure 13 shows the scheme. Between the two levels shown the four left-hand processors need to communicate with the four processors right below them and with the four right-hand processors on the right. The two loops shown serve this purpose. Each processor drops a data value into either the vertical connection loop or the cross-connection loop. This value is picked up by the target processor after 2^i ticks for the loops connecting levels i and $i+1$.

8 Conclusions

The optical computing technology has matured to the point that prototypical computing machines can be built in the research laboratories. This also is the right moment for the algorithm designers to get involved in this enterprise. A careful interaction between the architects and the algorithm designers can lead to a better-thought-out design. This paper is an attempt in that direction. We have studied the repercussions of the use of memory loops on algorithm design. The use of delay line loops as memories is necessitated by the required data rates (upward of 100 M words/s). We develop a computational model, the LMM, to capture the relevant characteristics of the memory loops.

It would seem that the restrictive discipline imposed on the data access patterns by a loop memory would degrade the performance of most algorithms, because the processor might have to idle waiting for data. We demonstrate that an important class of algorithms, ascend-descend algorithms, can be realized in the LMM without any loss of efficiency. In fact, the sequential realizations span a broad range for the number of loops required. A parallel implementation performing the optimal amount of work is also shown. Some matching lower bounds are illustrated, as well.

Optical computing is an emerging field. There are a myriad of open questions. We have only covered one class of algorithms in the LMM, the ascend-descend class. Many more problems and algorithms need to be investigated in this model. As we mentioned earlier, a large secondary memory that would be either semiconductor based or optical technology based would almost certainly be needed. In fact, the memory hierarchy is likely to have many more levels in this case due to the large mismatch in the speeds of the dynamic and secondary storage. A model similar to HMM (Ref. 13) to capture the hierarchical nature of memory would help.

The lower bounds in the LMM also pose many interesting problems. A new set of techniques seems to be required. The data access pattern of a problem has to be classified in terms of some basic frequencies. In summary, we believe that this field would serve as a fertile ground for future research.

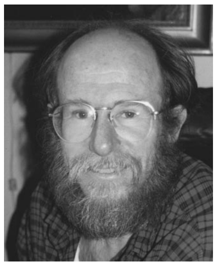
Acknowledgments

The research of J. Reif was supported in part by National Science Foundation (NSF) grant NSF-IRI-91-00681 and Army Research Office contract DAAH- 04-96-1-0448. A. Tyagi was supported by NSF Grant #MIP-8806169, NCBS&T Grant #89SE04, and a Junior Faculty Development award from the University of North Carolina, Chapel Hill. The support from these agencies is gratefully acknowledged.

References

1. H. T. Kung, ''Let's design algorithms for VLSI systems,''in *Proc. of the Caltech Conf. on Advanced Research in VLSI: Architecture, Design, Fabrication*, pp. 65–90 (Jan. 1979).
2. C. D. Thompson, ''Area-time complexity for VLSI'' in *Proc. ACM Symp. on Theory of Computing*, pp. 81–88, ACM-SIGACT (1979).
3. A. Huang, ''Design for an optical general purpose digital computer,'' in *Proc. Int. Optical Computing Conf., Proc. SPIE* **232**, 119–127 (1980).
4. A. Huang, ''Architectural considerations involved in the design of an optical digital computer,'' *Proc. IEEE* **72**(7), 780–786 (1984).
5. E. S. Maniloff, K. M. Johnson, and J. Reif, ''Holographic routing network for parallel processing machines,'' *Proc. SPIE* **1136**, 283 (Apr. 1989).
6. A. Louri, ''Three-dimensional optical architecture and data-parallel algorithms for massively parallel computing,'' *IEEE Micro.* **11**, 24–81 (Apr. 1991).
7. L. R. McAdams and J. W. Goodman, ''Liquid crystal $1\times n$ optical switch,'' *Opt. Lett.* **15**, 1150–1152 (1990).
8. L. R. McAdams, R. N. McRuer, and J. W. Goodman, ''Liquid crystal optical routing switch,'' *Appl. Opt.* **29**, 1304–1307 (1990).
9. A. Sawchuk and T. Strand, ''Digital optical computing,'' *Proc. IEEE* **72**(7), 758–779 (1984).
10. H. F. Jordan, ''Pipelined digital optical computing,'' Technical Report OCS Technical Report 89-34, Optoelectronic Computing Center, University of Colorado, Boulder (1989).
11. G. Q. Maguire and P. R. Prucnal, ''High-density optical storage using optical delay lines: use of optical delay lines as a disk,'' in *Medical Imaging III, Proc. SPIE* **1093**, 571–577 (Jan. 1989).
12. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA (1974).
13. A. Aggarwal, B. Alpern, A. Chandra, and M. Snir, ''A model for hierarchical memory,'' in *Proc. ACM Symp. on Theory of Computing*, pp. 305–314, ACM-SIGACT (1987).
14. A. Aggarwal, A. Chandra, and M. Snir, ''Hierarchical memory with block transfer,'' in *Proc. IEEE Symp. on Foundations of Computer Science*, pp. 204–216 (1987).
15. A. Aggarwal, A. Chandra, and M. Snir, ''On communication latency in PRAM computations,'' in *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pp. 11–21 (1989).
16. J. S. Vitter and E. A. M. Shriver, ''Optimal disk I/O with parallel block transfer,'' in *Proc. ACM Symp. on Theory of Computing*, pp. 159–169, ACM-SIGACT (1990).
17. R. Barakat and J. Reif, ''Lower bounds on the computational efficiency of optical computing systems,'' *Appl. Opt.* **26**, 1015–1018 (Mar. 1987).
18. A. Tyagi and J. Reif, ''Energy complexity of optical computations,'' in *Proc. 2nd IEEE Symp. on Parallel and Distributed Processing*, pp. 14–21 (Dec. 1990).
19. J. Reif and A. Tyagi, ''Efficient parallel algorithms for optical computing with the DFT primitive,'' in *Proc. 10th Conf. on Foundations of Software Technology & Theoretical Computer Science*, pp. 149–160, Lecture Notes in Computer Science #472, Springer-Verlag (Dec. 1990).
20. V. P. Heuring, H. F. Jordan, and J. Pratt, ''A bit-serial architecture for optical computing,'' *Appl. Opt.* **31**, 3213–3224 (1992).
21. H. F. Jordan, ''Digital optical computing with fibers and directional couplers,'' in *Optical Computing, OSA 1989 Technical Digest Series*, Vol. 9, *Optical Computing Topical Meeting*, pp. 352–355 (Feb. 1989).
22. H. F. Jordan, ''Digital optical computers at boulder,'' in *Optics for Computers: Architectures and Technologies, Proc. SPIE* **1505**, 87–98 (1991).
23. D. Lee, K. Y. Lee, and H. F. Jordan, ''Generalized lambda time-slot permuters,'' in *High-Speed Fiber Networks and Channels II, Proc. SPIE* **1784**, 262–269 (1993).
24. H. F. Jordan, K. Y. Lee, and D. Lee, ''Multichannel time-slot permuters,'' in *High-Speed Fiber Networks and Channels II, Proc. SPIE* **1784**, 252–261 (1993).
25. K. Y. Lee, D. Lee, and H. F. Jordan, ''Time-slot sorter,'' in *High-Speed Fiber Networks and Channels II, Proc. SPIE* **1784**, 242–251 (1993).
26. G. Barbarossa and P. J. Laybourn, ''Novel architecture for optical guided-wave recirculating delay lines,'' in *Advances in Optical Information Processing V, Proc. SPIE* **1704**, 138–144 (1992).
27. K. H. Wagner, S. Kraut, L. J. Griffiths, S. Weaver, R. T. Weverka, and A. W. Sarto, ''Efficient true-time-delay adaptive array processing,'' *Radar Processing, Technology, and Applications, Proc. SPIE* **2845**, 287–300 (1996).
28. D. Dolfi, J. Tabourel, O. Durand, V. Laude, J.-P. Huignard, and J. Chazelas, ''Optical architectures for programmable filtering of microwave signals,'' in *Radar Processing, Technology, and Applications, Proc. SPIE* **2845**, 276–286 (1996).
29. G. J. Chaitin, ''Register allocation and spilling via graph coloring,'' in *Proc. ACM Symp. on Compiler Construction*, pp. 98–105 (1982).
30. A. Tyagi, ''Reconfigurable memory queues/computing units architecture,'' in *Proc. Reconfigurable Architecture Workshop at 11th Int. Parallel Processing Symp.*, pp. 99–103 (Apr. 1997).
31. D. B. Sarrazin and H. F. Jordan, ''Delay line memory systems,'' Appl. Opt. **29**, 627–637 (1990) (also OCSC TR 92-18).
32. J. L. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan-Kaufmann, San Francisco, CA (1996).
33. J. D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, Rockville, MD (1984).
34. F. P. Preparata and J. Vuillemin, ''The cube-connected cycles: a versatile network for parallel computation,'' *Commun. ACM* **24**(5), 300–309 (1981).
35. P. Quinton, ''Automatic synthesis of systolic arrays from uniform recurrent equations,'' in *Proc. 11th Ann. Symp. on Computer Architecture*, pp. 208–214, IEEE (1984).
36. P. Quinton and V. van Dongen, ''The mapping of linear recurrence

equations on regular arrays,'' *J. VLSI Signal Process.* **1**, 95–113 (1989).
37. M. C. Chen, ''The generation of a class of multipliers: a synthesis approach to the design of highly parallel algorithms in VLSI,'' in *Proc. Int. Conf. on Computer Design*, pp. 116–121, IEEE (1985).
38. T. Lin and C. Mead, ''The application of group theory in classifying systolic arrays,'' Technical Report 5006:DF:82, California Institute of Technology, Pasadena (Apr. 1982).
39. H. S. Stone, ''Parallel processing with the perfect shuffle,'' *IEEE Trans. Comput.* **C-19**, 153–161 (Feb. 1970).
40. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA (1980).
41. F. C. Hennie, ''One-tape, off-line turing machine computations,'' *Inf. Control* **8**, 553–578 (1965).
42. F. C. Hennie, ''On-line turing machine computations,'' *IEEE Trans. Electron. Comput.* **EC-15**, 35–44 (1966).



John H. Reif received his BS degree (magna cum laude) from Tufts University in 1973 and his MS and PhD degrees from Harvard University in 1975 and 1977, respectively. He was a research assistant professor at the University of Rochester from 1978 to 1979, an assistant professor from 1979 to 1983, and associate professor from 1983 to 1986 at Harvard University. Since 1986 he has been a professor of computer science at Duke University. His research interests are theoretical computer science: efficient algorithms, randomized algorithms, parallel computation, robotics, data compression, and optical computing. He is a fellow of the Institute of Combinatorics, the Institute of Electrical and Electronics Engineers, and the Association for Computing Machinery.

Akhilesh Tyagi received a BE (honors) in electrical and electronics engineering from Birla Institute of Technology and Science, Pilani, India, in 1981, an MTech in computer engineering from Indian Institute of Technology, New Delhi, India, in 1983, and a PhD in computer science from University of Washington, Seattle, in 1988. He was with the Computer Science Department at University of North Carolina, Chapel Hill, until June 1993. He is currently with the Computer Science Department of Iowa State University, Ames. His research interests include very large scale integration (VLSI) complexity theory, CAD, and computer architecture.