Technical Report No. 99-431
Parallel Maximum Sum Algorithms on Interconnection Networks*

Ke Qiu
Jodrey School of Computer Science
Acadia University
Wolfville, Nova Scotia, Canada

Selim G. Akl
Dept. of Comp. & Info. Sci.
Queen's University
Kingston, Ontario, Canada

Abstract

We develop parallel algorithms for both one-dimensional and two-dimensional versions of the maximum sum problem (or max sum for short) on several interconnection networks. These algorithms are all based on a simple scheme that uses prefix sums. To this end, we first show how to compute prefix sums of N elements on a hypercube, a star, and a pancake interconnection network of size p (where $p \leq N$) in optimal time of $O(\frac{N}{p} + \log p)$. For the problem of maximum subsequence sum, the 1-D version of the max sum problem, we find an algorithm that computes the maximum sum of N elements on the aforementioned networks of size p, all with a running time of $O(\frac{N}{p} + \log p)$, which is optimal in view of the trivial $\Omega(\frac{N}{p} + \log p)$ lower bound. When $p = O(\frac{N}{\log N})$, our algorithm computes the max sum in $O(\log N)$ time, resulting in an optimal cost of $O(N)$. This result also matches the performance of two previous algorithms that are designed to run on PRAM.

Our 1-D max sum algorithm can be used to solve the problem of maximum subarray, the 2-D version of the general max sum problem. In particular, on the three interconnection networks mentioned above, our parallel algorithm finds the maximum subarray of an $N \times N$ array in time $O(\log N)$ with $O(\frac{N^3}{\log N})$ processors, once again, matching the performance of a previous PRAM algorithm. Note that the cost of the algorithm is $O(N^3)$, the same asymptotic time as the currently best known sequential algorithm.

Keywords: interconnection networks, hypercube, star graph, pancake graph, max subsequence sum problem, prefix sums.

1 Introduction

The problem of maximum sum has 1-D and 2-D versions. For the 1-D version, given a sequence of N numbers $x_0, x_1, ..., x_{N-1}$, the maximum subsequence sum (also referred to as the maximum sum subsequence sometimes) problem is to find two indices a and b, where $a \leq b$, such that $\sum_{i=a}^{b} x_i$ is the largest among all such subsequences. Obviously, the subsequence itself that leads to the maximum sum is not necessarily unique. For simplicity, we are only concerned with finding the maximum sum, instead of one actual subsequence that results in the sum. It is a simple matter of bookkeeping if we do want one actual subsequence. In the more general 2-D version, given an $N \times N$ array, we want to find a rectangular subarray with the maximum sum among all subarrays. The 2-D version has applications in pattern recognition and is also referred to as the maximum subarray problem.

Sequentially, the maximum subsequence sum problem can be solved in $O(N)$ time (e.g., see [3, 8, 13]). Typically, the sequential algorithm performs exactly one pass over the input and executes a constant number of constant-time operations for each x_i (see, for example, the elegant one-pass algorithm presented in [13]). Parallel algorithms have also been found for the PRAM model. In [10] and [14], it is shown that the problem can be solved on the EREW PRAM with $O(N/\log N)$ processors in $O(\log N)$ time. An ERCW PRAM algorithm, using a different approach but with the same performance, is given in [3]. An $O(1)$ time algorithm is also given in [3] that runs on a BSR model with N processors.

*This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC)

For the maximum subarray problem, an $O(N^3)$ sequential algorithm exists [6]. An $O(\log N)$ time PRAM parallel algorithm is given in both [10] and [14] using $O(N^3/\log N)$ processors.

In this paper, we develop several parallel algorithms for both versions of the max sum problem that run on interconnection networks, in particular, the hypercube, the star, and pancake networks. For the maximum subsequence sum problem, all the algorithms are optimal in view of the lower bounds that we establish on the corresponding networks (Note that they are optimal in the sense that no faster algorithm exists for the corresponding network because of the inherent constraint such as the network diameter and the number of elements held in each processor. This optimality is different from the usual cost-optimality, where the cost of a parallel algorithm is defined to be the product of time and the number of processors used) while some of them are actually cost-optimal. For the maximum subarray problem, the performance of our algorithm matches those of the ones given in [10] and [14] which run on a PRAM.

Because our parallel algorithm for solving the problem of maximum subarray on the given three interconnection networks is a natural extension of our algorithm for the maximum subsequence sum, emphasis will be placed on the latter. Consequently, we organize the paper as follows. In Section 2, we first present another $O(N)$ sequential algorithm based on the idea of prefix sums. This algorithm is the one on which all our parallel algorithms are based. In Section 3, we use hypercubes to show general schemes for computing prefix sums and the maximum subsequence sum. Specifically, we show that on an n-cube (where the number of processors p is 2^n):

1. on a fine-grained hypercube where each processor holds one input element, the maximum subsequence sum of a sequence of $N = p$ elements can be solved in $O(\log N)$ time (straightforward implementation of our sequential algorithm). This performance matches the trivial $\Omega(\log N)$ lower bound (which is the diameter of the hypercube with N nodes).

2. on a coarse-grained hypercube where each processor holds $O(N/p)$ elements, both the prefix sums and the maximum subsequence sum can be solved in $O(N/p + \log p)$ time, once again matching the $\Omega(N/p + \log p)$ lower bound that will be established.

Section 4 discusses how we can extend the algorithms to the star and pancake interconnection networks, two relatively new topologies for interconnecting processors in a parallel computer, so that the above two results also hold. Our results for the star and pancake graphs are interesting, considering the fact that, unlike a hypercube whose degree is logarithmic in terms of the total number of nodes, both stars and pancakes have sub-logarithmic degree. Section 5 discusses how we can solve the problem of maximum subarray on the networks using the results we developed earlier. Section 6 concludes the paper.

2 A Simple Sequential Algorithm

Given elements x_0, x_1, ..., x_{N-1}, and an associative binary operator $\oplus$, the prefix sums problem is to compute $s_j = \sum_{i=0}^{j} x_i$, for $0 \leq j \leq N-1$. For our purpose, we assume that the x_i's are numbers and the operator $\oplus$ is the regular addition operation.

If $\sum_{i=a}^{b} x_i$ is a maximum subsequence sum, where $0 \leq a \leq b \leq N-1$, then clearly $s_b - s_{a-1}$ is the largest among all $s_j - s_i$, for $-1 \leq i < j \leq N-1$, where s_{-1} is defined to be 0. This is because $\sum_{i=a}^{b} x_i = s_b - s_{a-1}$. This observation immediately leads to a maximum subsequence sum algorithm described below. The main idea is to find, for each prefix sum, the smallest prefix sum preceding it, then compute the difference between the two quantities. The largest of all these differences is the maximum sum.

1. Compute the prefix sums $s_j = \sum_{i=0}^{j} x_i$, for $0 \leq j \leq N-1$.

2. For each j, $0 \leq j \leq N-1$, find an index i_j, where $-1 \leq i_j \leq j-1$, such that s_{i_j} is the smallest among all such s_k's, $-1 \leq k \leq j-1$, where $s_{-1} = 0$. Namely, $s_{i_j} = \min_{-1 \leq k \leq j-1} s_k$.

3. Now that we have N pairs (s_0, s_{i_0}), (s_1, s_{i_1}), ..., $(s_{N-1}, s_{i_{N-1}})$, we can compute $\max_{0 \leq j \leq N-1}(s_j - s_{i_j})$, which is the maximum sum. If for some k, $s_k - s_{i_k} = \max_{0 \leq j \leq N-1}(s_j - s_{i_j})$, then one maximum subsequence is x_{i_k+1}, x_{i_k+2}, ..., x_k.

Clearly, the above algorithm can be implemented in linear time. One possible implementation is given as follows, where for any j, $0 \leq j \leq N-1$, current-min has the value $\min(s_{-1}, s_0, s_1, ..., s_{j-1})$ and $s_{-1} = 0$.

```
current-min ← 0
index ← -1
$s_{-1}$ ← 0
for $j = 0$ to $N - 1$ do
    $s_j \leftarrow s_{j-1} + x_j$
    $i_j$ ← index
    if $s_j$ < current-min then
        index ← $j$
        current-min ← $s_j$

MaxSubsequenceSum ← $\max_{0 \leq j \leq N-1}(s_j - s_{i_j})$.
```

This algorithm needs two **for** loops, one for finding the smallest prefix sum preceding any given prefix sum value, and one for finding the largest of all differences between a prefix sum and its corresponding smallest preceding prefix sum.

It is not hard to see that we can actually implement our algorithm in one pass, just like previous sequential algorithms, as follows. However, our subsequent parallel algorithms are based on the first version because it is easier to understand, more straightforward, and easier to parallelize.

```
current-max-sum ← $x_0$
current-min ← $\min(0, x_0)$
$s_0 \leftarrow x_0$
for $j = 1$ to $N - 1$ do
    $s_j \leftarrow s_{j-1} + x_j$
    current-max-sum ← max (current-max-sum, $s_j$ - current-min)
    current-min ← min (current-min, $s_j$)
```

where current-max-sum contains the maximum subsequence sum for the subsequence x_0, x_1, ..., x_j. When the algorithm terminates, variable current-max-sum has the value of the maximum sum for the original sequence.

3 A General Scheme: Solving Maximum Subsequence Sum Problem on a Hypercube

3.1 Maximum Subsequence Sum on a Fine-Grained Hypercube

Given sequence $x_0, x_1, ..., x_{N-1}$, our job is to find the maximum subsequence sum on a fine-grained hypercube of size $p = N$, where each processor (node) P_i holds one number x_i from the sequence.

A *hypercube of dimension* n, denoted Q_n, with p processors P_0, P_1, ..., P_{p-1}, where $p = 2^n$, is defined as follows. Let $i_{n-1}i_{n-2}\cdots i_{j+1}i_ji_{j-1}\cdots i_1i_0$ be i's binary representation. Processor $P_{i_{n-1}i_{n-2}\cdots i_{j+1}i_ji_{j-1}\cdots i_1i_0}$ is connected to processor $P_{i_{n-1}i_{n-2}\cdots i_{j+1}\bar{i_j}i_{j-1}\cdots i_1i_0}$ along dimension j for $0 \leq j < n$, where $\bar{i_j}$ is the complement of i_j. Therefore, each processor has n neighbors along dimensions 0, 1, ..., $n-1$. The n-dimensional hypercube is also referred to as an n-cube.

One of the many important properties of the n-cube is that it can be constructed recursively from lower dimensional cubes. More precisely, consider two identical $(n-1)$-cubes whose vertices are numbered likewise from 0 to $2^{n-1} - 1$. By joining every vertex of the first $(n-1)$-cube to the vertex of the second having the same number, one obtains an n-cube. Conversely, separating an n-cube into the subgraph of all the processors whose leading bit is 0 and the subgraph of all the processors whose leading bit is 1, the two subgraphs are such that each node of the first is connected to one node of the second along dimension $n-1$. If we remove the edges between these two graphs, we get two disjoint $(n-1)$-cubes. For convenience, we call the first Q_{n-1} the left sub-cube, and the second the right sub-cube. In this case, we say that the n-cube has been decomposed into two Q_{n-1}'s. Note that the splitting suggested above gives privilege to the leading bit but there is no particular reason for this. Since an n-cube has n dimensions, the splitting can be done along any of these n dimensions. A detailed treatment of hypercubes can be found in [9].

It can be easily seen that a trivial lower bound for the problem of computing the maximum sum on a hypercube is $\Omega(\log p) = \Omega(n)$ because $\log p$ is the diameter of the n-cube. It is straightforward to parallelize our sequential algorithm on the hypercube as follows.

Each processor P_i, $0 \leq i \leq N-1$, holds three variables s_i, v_{i_1}, and v_{i_2} where s_i is the prefix sum $\sum_{j=0}^{i} x_j$, $v_{i_1} = \min_{-1 \leq j \leq i-1} s_j$, the smallest prefix sum preceding s_i, and $v_{i_2} = \min_{-1 \leq j \leq i} s_j = \min(v_{i_1}, s_i)$. As we did earlier, define $s_{-1} = 0$. Clearly, the maximum subsequence sum is $\max_{0 \leq i \leq N-1}(s_i - v_{i_1})$.

Prefix sums can be computed on an n-cube in $O(\log p) = O(n)$ time easily (see, for example, [7]). Similarly, $\max_{0 \leq i \leq N-1}(s_i - v_{i_1})$ can be found in $O(\log p)$ time because each P_i can compute $s_i - v_{i_1}$ in $O(1)$ time and the maximum of these N differences can be found in $O(\log p)$ time. We now describe how to compute, for each i, v_{i_1} and v_{i_2}, assuming that prefix sums s_i's have been computed and stored in processor P_i's.

The algorithm falls into the ASCEND class that is standard for hypercube algorithms [11] in that we combine the results from two sub-cubes of dimension k to form the result for a sub-cube of dimension $k+1$, $k = 0, 1, ..., n-1$. Initially, for each $0 \leq i \leq N-1$, $v_{i_1} \leftarrow 0$, and $v_{i_2} \leftarrow \min(0, s_i)$. Assume that at some stage of the algorithm we have two sub-cubes of dimension k such that each processor P_i in the left sub-cube has a value for v_{i_1} and v_{i_2}, respectively. Also, the two sub-cubes are connected along links on dimension k. Let the processors in the left sub-cube be P_l, P_{l+1}, ..., P_{i-1}, P_i, ..., P_{l+2^k-1}, then v_{i_1} is the value of the smallest prefix sum among all prefix sums stored in the left sub-cube up to and including s_{i-1}: $v_{i_1} = \min_{l \leq j \leq i-1} s_j$, and v_{i_2} is the smallest prefix sum in the entire sub-cube: $v_{i_2} = \min_{l \leq j \leq l+2^k-1} s_j$. Let P_j be a processor in the right sub-cube who is a neighbor of P_i along dimension k, i.e., $j = i + 2^k$. The variables v_{j_1} and v_{j_2} are defined similarly. In order to combine the two k-cubes into a $(k+1)$-cube, processors in both sub-cubes communicate with their k-dimensional neighbors: $P_i \leftrightarrow P_{i+2^k}$.

1. P_i: $v_{i_2} \leftarrow \min(v_{i_2}, v_{j_2})$;

2. P_j: $v_{j_2} \leftarrow \min(v_{i_2}, v_{j_2})$;
   $v_{j_1} \leftarrow \min(v_{i_2}, v_{j_1})$;

This combining procedure takes constant time and is done exactly $n = \log p$ times (recall that $p = N$), starting from sub-cubes of size 0. When the algorithm terminates, it can be easily seen that for P_i, we have $v_{i_1} = \min_{0 \leq j \leq i-1} s_j$. The time for the entire computation for finding the maximum sum is therefore $O(\log p)$ $= O(\log N)$.

3.2 Maximum Subsequence Sum on a Coarse-Grained Hypercube

For ease of presentation and without loss of generality, we assume that N is a multiple of p in this section. In a coarse-grained hypercube, each processor P_i stores N/p elements $x_{i(N/p)}$, $x_{i(N/p)+1}$, ..., $x_{i(N/p)+((N/p)-1)}$.

Because of the facts that the diameter of the hypercube of size p is $\log p$ and that each processor has $O(N/p)$ elements, a trivial lower bound for the problem of computing the maximum subsequence sum on the hypercube is $\Omega(N/p + \log p)$. In the following, we present an algorithm whose performance matches this lower bound.

The algorithm for the coarse-grained hypercube can be stated as follows.

1. Compute prefix sums (so that P_i contains prefix sums $s_{i(N/p)}$, $s_{i(N/p)+1}$, ..., $s_{i(N/p)+((N/p)-1)}$. We denote these sums as s_{i_0}, s_{i_1}, ..., $s_{i_{(N/p)-1}}$, for ease of presentation.

2. Each processor has two variables v_{i_1} and v_{i_2} as defined earlier for the fine-grained hypercube algorithm. Initially, $v_{i_2} \leftarrow \min(0, s_{i_0}, s_{i_1}, ..., s_{i_{(N/p)-1}})$ and $v_{i_1} \leftarrow 0$.
   Run the fine-grained hypercube algorithm (with v_{i_1} and v_{i_2}) so that when it terminates, each processor knows v_{i_1}, the smallest prefix sum among all the prefix sums stored in processors P_0, P_1, ..., P_{i-1}, i.e., $v_{i_1} = \min(0, s_0, s_1, ..., s_{i_0 - 1})$.

3. Each processor P_i does the following:
   For each of its prefix sums s_{i_k}, where $0 \leq k \leq N/p - 1$, find $t_{i_k} \leftarrow \min_{0 \leq j \leq k-1}(0, s_{i_j})$. (This is similar

to our sequential maximum sum algorithm, where for each prefix sum we find the smallest preceding prefix sum residing in the same processor).
Then for each s_{i_k}, find $d_{i_k} \leftarrow \min(t_{i_k}, v_{i_1})$. That is, each prefix sum s_{i_k} knows its smallest preceding prefix sum $d_{i_k} = \min(0, s_0, s_1, ..., s_{i_k-1})$.
Now find $m_i \leftarrow \max(s_{i_0} - d_{i_0}, s_{i_1} - d_{i_1}, ..., s_{i_{\lceil N/p \rceil -1}} - d_{i_{\lceil N/p \rceil -1}})$.

4. Finally, maximum sum $\leftarrow \max_{0 \le i \le p-1} m_i$.

Both Steps 2 and 4 take $O(\log p)$ time while Step 3 needs $O(N/p)$ time. In the next subsection, we show that Step 1 needs no more time than $O(N/p + \log p)$ to compute prefix sums on a coarse-grained hypercube. Thus, the running time required for the coarse-grained maximum subsequence sum algorithm for the hypercube of size p is $O(N/p + \log p)$, optimal in view of the $\Omega(N/p + \log p)$ lower bound. When $p = N/\log N$), that is, each processor holds $\log N$ numbers, our algorithm runs in $O(\log N)$ time. Note that this result is the same as those of the algorithms in [10] and [14], which run on a PRAM.

3.2.1 Computing Parallel Prefix Sums

In this subsection, we discuss the problem of computing prefix sums on any coarse-grained interconnection network.

Given elements x_0, x_1, ..., x_{N-1}, stored in processors P_0, P_1, ..., P_{p-1} in a parallel computer with p processors, where $N \ge p$, such that P_i contains elements $x_{i(N/p)}$, $x_{i(N/p)+1}$, $x_{i(N/p)+2}$, ..., $x_{i(N/p)+((N/p)-1)}$, for $0 \le i \le p-1$, with the processors ordered such that $P_i \prec P_j$ if $i < j$, and an associative binary operator $\oplus$, the *parallel prefix* problem with respect to the processor ordering is to compute all the prefix sums $s_j = \sum_{i=0}^{j} x_i$, $0 \le j \le N-1$. Note that operation in $\sum$ is $\oplus$, not necessarily the regular arithmetic operator $+$. At the end of the computation we require that processor P_i contain $s_{i(N/p)}$, $s_{i(N/p)+1}$, $s_{i(N/p)+2}$, ..., $s_{i(N/p)+((N/p)-1)}$, for $0 \le i \le p-1$.

As usual, we assume that in $O(1)$ time, a processor can send or receive one datum to or from one of its neighbors. Under this assumption, a lower bound for the problem of computing prefix sums is $\Omega(N/p + \log p)$. To see this, computing N/p prefix sums within each processor takes $\Omega(N/p)$ time, while $\Omega(\log p)$ time is needed just to send any element in node P_0 to all the processors in the network in order to compute all prefix sums correctly since the problem of broadcasting one message on any network with P processors has a lower bound $\Omega(\log p)$ [2].

In what follows, we describe a generic parallel prefix algorithm for any parallel computer with P processors. The idea of the algorithm is simple. After Phase 1, each processor has the total sum of all the elements stored in that processor. In each step in Phase 2, processor P_i and P_{i+2^l} (in the last step, it is possible that processor P_{i+2^l} is non-existent) exchange their total sums. P_i "adds" the total sum just received to its own total sum and makes it the new total sum, while P_{i+2^l} does the same in addition to storing the total sum received from P_i in its memory. After Phase 2, each processor has all the necessary data to compute the final prefix sums. The algorithm is given as follows:

- Phase 1. For each processor P_i, $0 \le i \le p-1$, containing elements y_0, y_1, ..., $y_{N/p-1}$, compute in parallel $\sum_{j=0}^{k} y_j$, for $0 \le k \le N/p - 1$. Denote the total sum $\sum_{j=0}^{N/p-1} y_j$ by $\sum_i$.

- Phase 2. For l=0 step 1 until $\lceil \log p \rceil$-1:
  for each pair P_i and P_{i+2^l}, (that is, in the first step ($l = 0$) we pair P_0 with P_1, P_2 with P_3, and so on. In the second step ($l = 1$), we pair P_0 with P_2, P_1 with P_3, and so on. And in general, in the $l + 1^{st}$ step, P_0 is paired with P_{2^l}, P_1 is paired with P_{1+2^l}, P_2 is paired with P_{2+2^l}, and so on), P_i receives the total sum T_2 of P_{i+2^l}, P_{i+2^l} receives the total sum T_1 of P_i, both P_i and P_{i+2^l} replace their total sum with $T_1 \oplus T_2$, P_{i+2^l} also stores T_1 in its local memory for later use.

- Phase 3. Do in parallel, for $0 \le i \le p-1$, P_i computes total sums T of all stored values, and "adds" (using $\oplus$) it to the partial prefix sums already computed in Phase 1: $(\sum_{j=0}^{k} y_j) \oplus T$ for $0 \le k \le N/p-1$.

The following example illustrates the above algorithm where $p = 8$ and $N = 24$, $\sum_i$ denotes the total sum of all the elements stored in processor P_i, and $\sum_{i-j}$ denotes the total sum of all the elements stored

Table 1: The Initial Configuration

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
x_0	x_3	x_6	x_9	x_{12}	x_{15}	x_{18}	x_{21}
x_1	x_4	x_7	x_{10}	x_{13}	x_{16}	x_{19}	x_{22}
x_2	x_5	x_8	x_{11}	x_{14}	x_{17}	x_{20}	x_{23}

Table 2: Configuration After Phase 1

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
x_0 x_1 x_2	x_3 x_4 x_5	x_6 x_7 x_8	x_9 x_{10} x_{11}	x_{12} x_{13} x_{14}	x_{15} x_{16} x_{17}	x_{18} x_{19} x_{20}	x_{21} x_{22} x_{23}
$x_0 \oplus x_1$ $\sum_0 = \sum_{j=0}^{2} x_j$	$x_3 \oplus x_4$ $\sum_1 = \sum_{j=3}^{5} x_j$	$x_6 \oplus x_7$ $\sum_2 = \sum_{j=6}^{8} x_j$	$x_9 \oplus x_{10}$ $\sum_3 = \sum_{j=9}^{11} x_j$	$x_{12} \oplus x_{13}$ $\sum_4 = \sum_{j=12}^{14} x_j$	$x_{15} \oplus x_{16}$ $\sum_5 = \sum_{j=15}^{17} x_j$	$x_{18} \oplus x_{19}$ $\sum_6 = \sum_{j=18}^{20} x_j$	$x_{21} \oplus x_{22}$ $\sum_7 = \sum_{j=21}^{23} x_j$

in processors P_i, P_{i+1}, ..., P_j. Table 1 shows the initial configuration. The status of each processor after Phase 1 is shown in Table 2. The purpose of Phase 2 is to compute all data needed to compute prefix sums correctly. For example, in Step 2, Processor P_1 receives $\sum_{2-3}$ from P_3, "adds" $\sum_{2-3}$ to its old total sum $\sum_{0-1}$ it already has, and stores the new sum $\sum_{0-3}$ in place of the old total sum, while P_3 receives $\sum_{0-1}$ from P_1, stores it, and "adds" it to $\sum_{2-3}$ it already has to get $\sum_{0-3}$, which is stored as its new total sum. The contents of each processor after Steps 1, 2, and 3 of Phase 2 are given in Tables 3, 4, and 5, respectively. After Phase 2, each processor has all necessary data to compute the final result. For example, P_7 now contains x_{21}, x_{22}, x_{23}, $x_{21} \oplus x_{22}$, $\sum_{0-3}$, $\sum_{4-5}$, $\sum_6$, and $\sum_7$. Then in Phase 3, all P_7 has to do is to compute $\sum_{0-3} \oplus \sum_{4-5} \oplus \sum_6$ and "add" it to x_{21}, $x_{21} \oplus x_{22}$, and $x_{21} \oplus x_{22} \oplus x_{23}$ (this is $\sum_7$). The contents of all the processors are shown in Table 6, which are all required prefix sums.

As for the time complexity, Phase 1 takes $O(N/p)$ time as there are N/p elements stored in each processor. Phase 2 takes $O(C_{comm} \log p)$ time, where C_{comm} is the cost of transmitting total sums between two processors in each step in the given communication pattern (ASCEND) in Phase 2. Phase 3 requires $O(N/p + \log p)$ time where $O(\log p)$ is used to add all necessary data (there are at most $\lceil \log p \rceil$ of them) obtained from Phase 2 and $O(N/p)$ is used to add this sum to each of N/p prefix sums computed in Phase 1. The total time required is therefore $O(N/p + C_{comm} \log p)$.

For a hypercube of any size, this $C_{comm} = O(1)$. Therefore, the total time required to compute prefix sums of N elements on a hypercube of size p is $O(N/p + \log p)$. This results means that when $p = N/\log N)$, and each processor holds $\log N$ numbers, the prefix sums can be computed in $O(\log N)$ time, resulting in an

Table 3: Configuration After Step 1 of Phase 2

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
x_0 x_1 x_2	x_3 x_4 x_5	x_6 x_7 x_8	x_9 x_{10} x_{11}	x_{12} x_{13} x_{14}	x_{15} x_{16} x_{17}	x_{18} x_{19} x_{20}	x_{21} x_{22} x_{23}
$x_0 \oplus x_1$ $\sum_0$ $\sum_{0-1}$	$x_3 \oplus x_4$ $\sum_1$ $\sum_0$ $\sum_{0-1}$	$x_6 \oplus x_7$ $\sum_2$ $\sum_{2-3}$	$x_9 \oplus x_{10}$ $\sum_3$ $\sum_2$ $\sum_{2-3}$	$x_{12} \oplus x_{13}$ $\sum_4$ $\sum_{4-5}$	$x_{15} \oplus x_{16}$ $\sum_5$ $\sum_4$ $\sum_{4-5}$	$x_{18} \oplus x_{19}$ $\sum_6$ $\sum_{6-7}$	$x_{21} \oplus x_{22}$ $\sum_7$ $\sum_6$ $\sum_{6-7}$

Table 4: Configuration After Step 2 of Phase 2

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
x_0	x_3	x_6	x_9	x_{12}	x_{15}	x_{18}	x_{21}
x_1	x_4	x_7	x_{10}	x_{13}	x_{16}	x_{19}	x_{22}
x_2	x_5	x_8	x_{11}	x_{14}	x_{17}	x_{20}	x_{23}
$x_0 \oplus x_1$	$x_3 \oplus x_4$	$x_6 \oplus x_7$	$x_9 \oplus x_{10}$	$x_{12} \oplus x_{13}$	$x_{15} \oplus x_{16}$	$x_{18} \oplus x_{19}$	$x_{21} \oplus x_{22}$
$\sum_0$	$\sum_1$	$\sum_2$	$\sum_3$	$\sum_4$	$\sum_5$	$\sum_6$	$\sum_7$
$\sum_{0-3}$	$\sum_0$	$\sum_{0-1}$	$\sum_2$	$\sum_{4-7}$	$\sum_4$	$\sum_{4-5}$	$\sum_6$
	$\sum_{0-3}$	$\sum_{0-3}$	$\sum_{0-1}$		$\sum_{4-7}$	$\sum_{4-7}$	$\sum_{4-5}$
			$\sum_{0-3}$				$\sum_{4-7}$

Table 5: Configuration After Step 3 of Phase 2

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
x_0	x_3	x_6	x_9	x_{12}	x_{15}	x_{18}	x_{21}
x_1	x_4	x_7	x_{10}	x_{13}	x_{16}	x_{19}	x_{22}
x_2	x_5	x_8	x_{11}	x_{14}	x_{17}	x_{20}	x_{23}
$x_0 \oplus x_1$	$x_3 \oplus x_4$	$x_6 \oplus x_7$	$x_9 \oplus x_{10}$	$x_{12} \oplus x_{13}$	$x_{15} \oplus x_{16}$	$x_{18} \oplus x_{19}$	$x_{21} \oplus x_{22}$
$\sum_0$	$\sum_1$	$\sum_2$	$\sum_3$	$\sum_4$	$\sum_5$	$\sum_6$	$\sum_7$
$\sum_{0-7}$	$\sum_0$	$\sum_{0-1}$	$\sum_2$	$\sum_{0-3}$	$\sum_4$	$\sum_{0-3}$	$\sum_6$
	$\sum_{0-7}$	$\sum_{0-7}$	$\sum_{0-1}$	$\sum_{0-7}$	$\sum_{0-3}$	$\sum_{4-5}$	$\sum_{0-3}$
			$\sum_{0-7}$		$\sum_{0-7}$	$\sum_{0-7}$	$\sum_{4-5}$
							$\sum_{0-7}$

Table 6: Configuration After Phase 3, The Final Result

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
x_0	x_3	x_6	x_9	x_{12}	x_{15}	x_{18}	x_{21}
x_1	x_4	x_7	x_{10}	x_{13}	x_{16}	x_{19}	x_{22}
x_2	x_5	x_8	x_{11}	x_{14}	x_{17}	x_{20}	x_{23}
$\sum_{j=0}^{0} x_j$	$\sum_{j=0}^{3} x_j$	$\sum_{j=0}^{6} x_j$	$\sum_{j=0}^{9} x_j$	$\sum_{j=0}^{12} x_j$	$\sum_{j=0}^{15} x_j$	$\sum_{j=0}^{18} x_j$	$\sum_{j=0}^{21} x_j$
$\sum_{j=0}^{1} x_j$	$\sum_{j=0}^{4} x_j$	$\sum_{j=0}^{7} x_j$	$\sum_{j=0}^{10} x_j$	$\sum_{j=0}^{13} x_j$	$\sum_{j=0}^{16} x_j$	$\sum_{j=0}^{19} x_j$	$\sum_{j=0}^{22} x_j$
$\sum_{j=0}^{2} x_j$	$\sum_{j=0}^{5} x_j$	$\sum_{j=0}^{8} x_j$	$\sum_{j=0}^{11} x_j$	$\sum_{j=0}^{14} x_j$	$\sum_{j=0}^{17} x_j$	$\sum_{j=0}^{20} x_j$	$\sum_{j=0}^{23} x_j$

optimal cost of $O(N)$.

4 Extensions to Stars and Pancakes

The star and pancake networks were proposed to be attractive alternatives to the hypercube topology for interconnecting many processors in a parallel computer, and compare favorably with it in several aspects [1, 2]. For example, an n-star or n-pancake has $p = n!$ nodes, but both their degree and diameter are $O(n)$, i.e., sub-logarithmic in the number of vertices, while a hypercube with $O(n!)$ vertices has a degree and diameter of $O(\log(p)) = O(n \log n)$, i.e., logarithmic in the number of vertices.

In this section, we study the problems of computing parallel prefix sums and finding the maximum subsequence sum on the star and pancake networks. Specifically, parallel algorithms are presented which solve both problems for an input sequence of N elements on a star or pancake network with p processors in time $O(N/p + \log p)$, where $N \geq p$. Both algorithms are optimal in view of the $\Omega(N/p + \log p)$ lower bound. On a fine-grained model where $N = p$, the time becomes $O(\log p)$, also optimal.

We will first introduce the two networks. We then see how our general schemes developed in Section 3 can be extended to run on these two networks, using a special routing algorithm.

Let V_n be the set of all $n!$ permutations of symbols 1, 2, ..., n. For any permutation $v \in V_n$, if we denote the i^{th} symbol of v by $v(i)$, then v can be written as $v(1)v(2)\cdots v(n)$. A *star interconnection network* on n symbols, $S_n = (V_n, E_{S_n})$, is an undirected graph with $n!$ vertices, where each vertex v is connected to $n-1$ vertices which can be obtained by interchanging the first and i^{th} symbols of v, i.e., $(v(1)v(2)\cdots v(i-1)v(i)v(i+1)\cdots v(n), v(i)v(2)\cdots v(i-1)v(1)v(i+1)\cdots v(n)) \in E_{S_n}$, for $2 \leq i \leq n$. We call these $n-1$ connections *dimensions*. Thus each vertex is connected to $n-1$ vertices through dimensions 2, 3, ..., n. S_n is also called an n-star. Fig. 1 shows S_4.

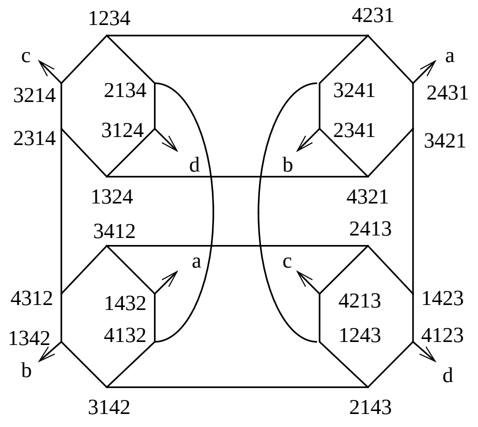

Figure 1: A 4-Star S_4

A *pancake interconnection network* on n symbols, $P_n = (V_n, E_{P_n})$, is an undirected graph with $n!$ vertices, where each vertex v is connected to $n-1$ vertices which can be obtained by flipping the first i symbols of v, i.e., $(v(1)v(2)\cdots v(i-1)v(i)v(i+1)\cdots v(n), v(i)v(i-1)\cdots v(2)v(1)v(i+1)\cdots v(n)) \in E_{P_n}$, for $2 \leq i \leq n$. The dimensions for P_n are defined similarly. P_n is also called an n-pancake. Fig. 2 shows P_4. Both S_n and P_n are in the family of Cayley graphs [1]. Since the following discussion and results apply to both networks, we henceforth use X_n to denote either S_n or P_n.

Let m_1 and m_2 be two distinct symbols from $\{1, 2, ..., n\}$. We use the notation $m_1 * m_2$ to represent a permutation of $\{1, 2, ..., n\}$ whose first and last symbols are m_1 and m_2, respectively, and * represents any permutation of the $n-2$ symbols in $\{1, 2, ..., n\} - \{m_1, m_2\}$. Similarly, m_1* is a permutation of n symbols whose first symbol is m_1, and $*m_2$ is a permutation of n symbols whose last symbol is m_2.

Let $X_{n-1}(i)$ be a sub-graph of X_n induced by all the vertices with the same last symbol i, for some $1 \leq i \leq n$. It can be seen that $S_{n-1}(i)$ is an $(n-1)$-star and that $P_{n-1}(i)$ is an $(n-1)$-pancake, both defined on symbols $\{1, 2, \cdots, n\} - \{i\}$. It follows that X_n can be decomposed into n X_{n-1}'s: $X_{n-1}(i)$, $1 \leq i \leq n$

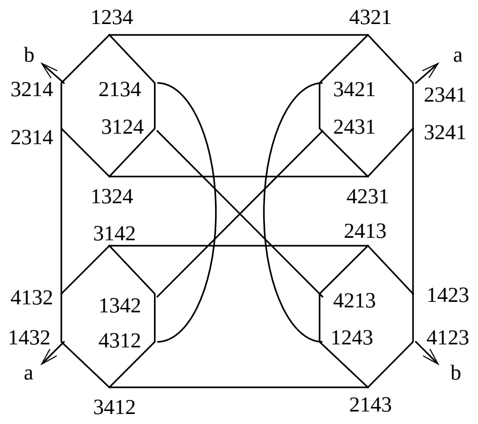

Figure 2: A 4-Pancake P_4

[1, 2]. For example, S_4 in Fig. 1 contains four 3-stars, namely $S_3(1)$, $S_3(2)$, $S_3(3)$, and $S_3(4)$, by fixing the last symbol at 1, 2, 3, and 4, respectively. P_n can also be decomposed similarly.

4.1 Parallel Maximum Subsequence Sum Algorithms on Stars and Pancakes

Even though X_n has a diameter of $O(n) = o(\log p)$, it is not hard to see that a lower bound for both the prefix sums and the max sum algorithms is still $\Omega(\log p) = O(n \log n)$ on a fine-grained model, and $\Omega(N/p + \log p)$ on a coarse-grained model. This is so simply because just to add all p numbers (with each processor holding one number) requires a time of at least $\Omega(\log p)$.

As mentioned earlier, most hypercube algorithms are of the type ASCEND or DESCEND, or some variation thereof [11]. Those hypercube algorithms that fall in this class can be further divided into two sub-classes according to whether or not they preserve the ordering of the processors while ascending or descending, i.e., whether or not a processor ranked k^{th} in one sub-cube has to communicate with a processor also ranked k^{th} in another sub-cube. Examples of algorithms in the first sub-class are Batcher's bitonic merging and sorting algorithms [5] as implemented on the hypercube. Algorithms that belong to the second sub-class include broadcasting and prefix sums computation. We will describe a routing algorithm called GROUP-COPY later which allows us to route data among different sub-stars and pancakes in a certain fashion (although not order-preserving) in constant time. This routing algorithm allows one to perform recursive doubling/halving [3] on X_n easily. The general scheme G for recursive doubling on X_n can be stated as follows, where G is whatever the algorithm being used to solve a problem.

```
$G(X_n)$
    **for** all $i$, $1 \leq i \leq n$, **do in parallel** $G(X_{n-1}(i))$
    **for** $i = 1$ **to** $\lceil \log n \rceil$ **do**
            starting from $X_{n-1}(1)$, arrange all $X_{n-1}$'s into groups of $2^i$ consecutive $X_{n-1}$'s.

            within each group are two subgroups, each with $2^{i-1}$ $X_{n-1}$'s, these two
            subgroups communicate with each other using the routing
            GROUP-COPY in order to do necessary work.
```

Of course, care must be taken as n is not necessarily a power of 2. The time required is $t(n) = t(n-1) + O((C_{comm} + C_{work}) \log n)$ with the solution $t(n) = O((C_{comm} + C_{work}) n \log n) = O((C_{comm} + C_{work}) \log p)$, where C_{comm} and C_{work} are costs for communication and work, respectively. Later on we will see that $C_{comm} = O(1)$. Also, since only constant number of operations are performed after each communication step (normally a few arithmetic operations such as addition) for all of our algorithms, $C_{work} = O(1)$. It is obvious that using the routing and the above general scheme, we can directly implement on the star and pancake networks any algorithm for the hypercube that is in the second sub-class, using asymptotically the same number of processors and with no asymptotic time loss. Note that this is an improvement since the star and pancake networks are weaker computational models compared with the hypercube (due to their smaller degree). The prefix sums algorithm in the previous subsection is one such typical example. There are a host of other basic data communication algorithm that also fall into the the second sub-class, including prefix sums computation and finding max/min [12].

In order to extend all of our hypercube maximum subsequence sum algorithms (both fine and coarse-grained models), all we need to do is to be able to compute prefix sums (both models), to find maximum of elements stored in X_n, and to do general recursive doubling. All of the above can be done using our general scheme and the routing algorithm.

We now briefly describe how prefix sums computations can be done in times $O(\log p) = O(n \log n)$ (fine-grained) and $O(N/p + \log p)$ (coarse-grained), respectively.

We first define the processor ordering. In X_n, let r denote the processor associated with the vertex $a_1a_2...a_n$ and q denote the processor associated with the vertex $b_1b_2...b_n$. The ordering, $\prec$, on the processors is defined as follows: $r \prec q$ if there exists an i, $1 \leq i \leq n$, such that $a_j = b_j$ for $j > i$, and $a_i < b_i$. In other words, the processors are ordered in reverse lexicographic order (i.e., lexicographic order if we read from right to left). If $r \prec q$, we say that r *precedes* q. The *rank* $r(u)$ of a vertex u is the number of vertices v such that $v \prec u$, i.e., $\mathrm{r}(u) = |\{v | v \prec u, v \in V_n\}|$. Clearly, $0 \leq r(u) \leq n! - 1$.

By this definition, we can write the $n!$ processors in X_n as P_i, $0 \leq i \leq n! - 1$, such that $P_i \prec P_j$ if $i \leq j$. For example, the six nodes of X_3 are labelled as follows:

$$P_0 = 321,\ P_1 = 231,\ P_2 = 312,\ P_3 = 132,\ P_4 = 213,\ P_5 = 123$$

We first take a look at the fine-grained version. Suppose that we have computed prefix sums for two groups of sub-structures as follows:

$$\begin{array}{l} \mathit{Group}\ 1 : X_{n-1}(i) \cdots\cdots\cdots X_{n-1}(i+k) \\ \mathit{Group}\ 2 : X_{n-1}(i+k+1) \cdots X_{n-1}(i+2k+1) \end{array}$$

and that each processor holds two variables s and t, for storing the partial prefix sum computed so far and the total sum of values in the group it is in, respectively. Let the total sum in Group 1 be t_1 and the total sum in Group 2 be t_2. We first use GROUP-COPY (the description of which is given next) to send t_1 to every processor in Group 2, and t_2 to every processor in Group 1, then the prefix sums in processors in Group 1 remain the same, while the prefix sum s in a processor in Group 2 becomes $s \oplus t_1$. The total sum for all the processors in both groups becomes $t_1 \oplus t_2$. All these steps can be accomplished in $O(1)$ time. When a group contains only one X_{n-1}, the algorithm is called recursively. Let $t(n)$ be the time complexity of the algorithm, then $t(n) = t(n-1) + O(\log n) = O(n \log n) = O(\log p)$, which is optimal.

The maximum and the minimum of $n!$ values stored one per node in X_n can also be found in $O(n \log n)$ $= O(\log p)$ time by letting the binary associative operation in the prefix sums algorithm be *max* and *min*, respectively.

The implementation of the coarse-grained hypercube prefix sums algorithm on X_n is straightforward. In Phase 1 each processor in X_n simply computes the prefix sums of its own elements. In Phase 2, we first recursively run Phase 2 in each $X_{n-1}(i)$ in parallel, for $1 \leq i \leq n$, then all that remains to be done is to treat each $X_{n-1}(i)$ as one processor (since all the processors in $X_{n-1}(i)$ contain the same set of data with the exception of their original elements) and execute each step of Phase 2. Phase 3 on X_n is exactly the same as before.

As for the running time of the coarse-grained parallel prefix sums algorithm on X_n, Phase 1 takes $O(N/p)$ time. Let $t(n)$ be the time required to implement Phase 2 on X_n, then

$$\begin{aligned} t(n) &= t(n-1) + O(C_{comm} \lceil \log n \rceil) \\ &= O(C_{comm} \log(n!)). \end{aligned}$$

By our routing algorithm, C_{comm} is a constant, thus $t(n) = O(\log(n!)) = O(\log p)$. Phase 3 takes $O(N/p + \log p)$ time for the same reason given before. Therefore, the total time required to compute all the prefix sums on X_n is $O(N/p + \log p)$, matching the lower bound.

The key routing algorithm used is an algorithm we developed in [4] that can route data between any two groups of S_{n-1}'s. This routing algorithm can be described as follows. Consider the following problem: Given $X_{n-1}(i)$, and $X_{n-1}(j)$, with $i \neq j$, it is required to exchange the contents of the processors in $X_{n-1}(i)$ with the processors in $X_{n-1}(j)$. By "exchanging the contents of $X_{n-1}(i)$ with $X_{n-1}(j)$" we mean that the content of each processor in $X_{n-1}(i)$ is exchanged with the content of a processor in $X_{n-1}(j)$ such that no two processors in $X_{n-1}(i)$ exchange their contents with the same processor in $X_{n-1}(j)$. This task can be accomplished in $O(1)$ time by a procedure called COPY [4]:

```
Procedure COPY $(i, j)$

1. do in parallel for all vertices $*i$, $*j$
   send content to neighbor along dimension $n$

2. do in parallel for all edges $(i * k, j * k)$, such that $k \neq i, j$,
   exchange contents between $i * k$ and $j * k$.

3. do in parallel for all vertices $i * k$, $j * k$, $k \neq i, j$
   send content to neighbor along dimension $n$. □
```

The above routing can be extended as follows. Let I: $i_1, i_2, ..., i_l$ and J: $j_1, j_2, ..., j_l$ be two sequences from $\{1, 2, ..., n\}$ such that no two elements of I are equal, no two elements of J are equal, and $\{i_1, i_2, ..., i_l\} \cap \{j_1, j_2, ..., j_l\} = \emptyset$. It is desired to exchange the contents of $X_{n-1}(i_m)$ with $X_{n-1}(j_m)$, in parallel, for $1 \leq m \leq l$. This task can also be achieved in constant time by a procedure called GROUP-COPY [4].

With the above algorithms (the general scheme for recursive doubling, prefix sums, and finding max/min), it is not hard to see that all of our hypercube algorithms for the maximum subsequence sum can be extended to run on both S_n and P_n without asymptotic time loss. That is, on a fine grained X_n, the running time is $O(\log p)$, and on a coarse-grained X_n, it is $O(N/p + \log p)$, both optimal.

5 Parallel Algorithm for Finding Maximum Subarray

A parallel algorithm to solve the problem of maximum subarray can be easily found using prefix sums computation and our our maximum subsequence sum algorithm already developed. The idea of this algorithm is from [10] and is outlined below.

Given matrix $A = (a_{ij})_{N \times N}$, for each pair (i, j), where $1 \leq i \leq j \leq N$, we find the maximum subarray with max value S_{ij}, i.e., $S_{ij} = \max_{1 \leq l \leq m \leq N} \sum_{s=i}^{j} \sum_{t=l}^{m} a_{st}$. We then find the largest one among all such sums $\max 1 \leq i \leq j \leq N S_{ij}$, which is the sum of the maximum subarray in the original $N \times N$ array.

One possible implementation is as follows. Initially, N processors are assigned for each column k and the prefix sums of these N numbers a_{1k}, a_{2k}, ..., a_{Nk} in column k can be found in $O(\log N)$ time. The sum of elements on column k that are between rows i and j $s_{ijk} = a_{ik} + a_{i+1,k} + ... + a_{jk}$ is then simply the difference between 2 prefix sums $(a_{1k} + ... + a_{jk})$ - $(a_{1k} + ... + a_{i-1,k})$, which can be computed in constant time. Now, for the given i and j, we have N sums s_{ij1}, s_{ij2}, ..., s_{ijN}, and the maximum subarray S_{ij} is simply the maximum subsequence of the sequence s_{ij1}, s_{ij2}, ..., s_{ijN}, which can be obtained in $O(\log N)$ time with N processors.

Note that prefix sums of each column needs to be computed only once, requiring $O(\log N)$ time and N^2 processors. Finding S_{ij} requires N processors and $O(\log N)$ time. Since we have to execute the above procedure for each of $O(N^2)$ pairs of i's and j's, we have an algorithm that uses $O(N^3)$ processors to solve the maximum subarray in $O(\log N)$ time. This algorithm has a cost of $O(N^3 \log N)$.

An improvement can be made immediately because (1) prefix sums of N numbers can be computed in $O(\log N)$ time with $p = N/\log N$ processors; and (2) maximum subsequence sum problem can be solved in $O(\log N)$ time, also with $p = N/\log N$ processors. This implies that the problem of maximum subarray can be solved in $O(\log N)$ time with $O(N^2 \times N/\log N)$ processors. The cost now becomes $O(N^3)$, the time required for the currently best known sequential algorithm. In addition, this result matches those of the PRAM algorithms given in [10] and [14]. It is trivial to see that the above algorithm works on PRAM as well as hypercubes, stars, and pancakes.

Similar to what we did earlier, we can make our algorithm more general in that (1) p processors can be assigned to each column when computing prefix sums initially; and (2) p processors can be used to solve the maximum subsequence problem for each pair (i, j), all with a time of $O(N/p + \log p)$. Therefore, the algorithm runs in time $O(N/p + \log p)$ with $O(N^2 p)$ processors.

6 Conclusion

Algorithms are developed to solve the maximum sum problem on several interconnection networks. These algorithms are all based on a simple linear sequential algorithm that uses prefix sums. We first showed how to compute prefix sums of N elements on a hypercube, a star, and a pancake interconnection network of size p (where $p \leq N$) in optimal time of $O(\frac{N}{p} + \log p)$. We then found algorithms that compute the maximum subsequence sum of N elements on the aforementioned networks of size p, all with a running time of $O(\frac{N}{p} + \log p)$, optimal in view of the trivial $\Omega(\frac{N}{p} + \log p)$ lower bound. For the fine-grained model where each processor holds one element, the algorithms for all three interconnection networks have a time complexity of $O(\log p) = O(\log N)$, also optimal. Our results for the star and pancake graphs are interesting, considering the fact that, unlike a hypercube whose degree is logarithmic in terms of the total number of nodes, both stars and pancakes have sub-logarithmic degree. In addition, when $p = N/\log N$, the problem can be solved in $O(\log N)$ time. The cost of the algorithm is therefore $O(N)$, which is optimal. Also, this performance matches that of a previous algorithm which is designed to run on a PRAM machine.

For the problem of maximum subarray, our algorithm for the three interconnection networks has a running time of $O(\log N)$ using $O(N^3/\log N)$ processors, which, once again, matches the results given in [10] and [14], which are for a much stronger PRAM.

References

[1] S.B. Akers and B. Krishnamurthy, "A Group Theoretic Model for Symmetric Interconnection Networks," *IEEE Transaction on Computers*, Vol. c-38, No. 4, April 1989, pp. 555-566.

[2] S.B. Akers, D. Harel, and B. Krishnamurthy, "The Star Graph: An Attractive Alternative to the n-cube," *Proc. International Conference on Parallel Processing*, St. Charles, Illinois, August 1987, pp. 393-400.

[3] S.G. Akl, *Parallel Computation: Models and Methods*, Prentice Hall, Upper Saddle River, New Jersey, 1997.

[4] S.G. Akl and K. Qiu, "A Novel Routing Scheme on the Star and Pancake Networks and Its Applications," *Parallel Computing*, **19**, 1, 1993, pp. 95-101.

[5] K.E. Batcher, "Sorting Networks and Their Applications," *AFIPS Conf. Proc. SJCC*, 32, Washington, D.C., Thompson Books, 1968, pp. 307-314.

[6] J. Bentley, "Programming Pearls: Solutions for September's Problem," *Communications of the ACM*, 27, Nov. 1984, pp. 1090-1092.

[7] M. Cosnard and D. Trystram, *Parallel Algorithms and Architectures*, International Thomson Computer Press, London, 1995.

[8] D. Gries, "A Note on the Standard Strategy for Developing Loop Invariants and Loops," *Sci. Compu. Programming*, 2, 1982, pp. 207-214.

[9] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufman Publishers, San Mateo, California, 1992.

[10] K. Perumalla and N. Deo, "Parallel Algorithms for Maximum Subsequence and Maximum Subarray," *Parallel Processing Letters*, Vol. 5, 1995, pp. 367-373.

[11] F.P. Preparata and J. Vuillemin, “The Cube-Connected Cycles: A Versatile Network for Parallel Computation,” *Communications of the ACM*, Vol. 24, May 1981, pp. 300-309.

[12] K. Qiu, S.G. Akl, and H. Meijer, “On Some Properties and Algorithms on the Star and Pancake Interconnection Networks,” *Journal of Parallel and Distributed Computing*, **22**, 1994, pp. 16-25.

[13] M.A. Weiss, *Data Structures and Algorithm Analysis*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[14] Z. Wen, “Fast Parallel Algorithms for the Maximum Sum Problem,” *Parallel Computing*, 21, 1995, pp. 461-466.