An Efficient Algorithm for the Nearest Smaller Problem on Distributed Shared Memory Systems with Applications *

T. Graf  
Department ICG-4  
Research Centre Jülich  
52425 Jülich, Germany  
t.graf@kfa-juelich.de

V. Kamakoti  
N. Balakrishnan  
Laboratory for High Performance Computing  
Supercomputer Education and Research Centre  
Indian Institute of Science  
Bangalore - 560 012, India  
{ kama, balki }@serc.iisc.ernet.in

Abstract

We present a simple and efficient algorithm for the nearest smaller problem (NSP, [1]) on a distributed shared memory (DSM) system with applications to problems from diverse areas. We adopt the block distributed memory (BDM) model of computation as described in [2]. To the best of our knowledge this is the first known algorithm for the NSP on DSM systems. Since the NSP is fundamental in many problems, a solution for it on DSM systems implies DSM-based solutions for a variety of problems in diverse areas as discussed in this paper. Parallel algorithms known so far for the NSP are based on shared memory systems [1] and are therefore less scalable than our algorithm.

1. Introduction

1.1. Motivation

The nearest smaller problem (NSP) is a fundamental problem and finds extensive applications in merging sorted lists, triangulation, binary tree reconstruction, parenthesis matching etc., [1].

Based on their memory organization, parallel computing systems fall into two categories: shared memory systems and distributed memory systems. Shared memory systems are relatively easy to program (due to a single address space) but less scalable than distributed memory systems. Configuring a distributed memory system to have a single address space results in a system that is both scalable and easy to program.

Such systems are called scalable shared memory systems or distributed shared memory systems (DSM). In other words, a DSM system is a shared memory layer on top of any distributed memory system like IBM's SP2, CRAY's T3E, or a cluster of workstations. In [2] the block distributed memory (BDM) model of computation is presented which serves as a bridge between the shared memory programming model and the distributed memory message-passing architecture. In other words, the BDM model attempts to capture the performance of a DSM system. So far, all parallel algorithms for the NSP are based on shared memory systems; [1] presents a $\frac{n}{\log n}$ processor, $O(\log n)$ time CREW-PRAM parallel algorithm and a $\frac{n}{\log\log n}$ processor, $O(\log\log n)$ time CRCW-PRAM parallel algorithm for the NSP. This paper presents a simple and efficient algorithm for the NSP on DSM systems using the BDM model. To the best of our knowledge, this is the only reported algorithm for the NSP on DSM systems.

Definition 1 (Nearest Smaller [1]) The input to this problem is an array $A = (a_1, a_2, \ldots, a_n)$ of $n$ elements from a totally ordered domain. For each $a_i$, $1 \leq i \leq n$, find the nearest element to its left and the nearest element to its right, that are less than $a_i$, if such elements exist. That is, for each $1 \leq i \leq n$, find the maximal $1 \leq j < i$, and the minimal $i < k \leq n$ such that $a_j < a_i$ and $a_k < a_i$. We say that $a_j$ is the left match and $a_k$ is the right match of $a_i$.

In the rest of this paper we will concentrate on finding the left match for every element of a given input sequence. Finding the right match can be done in a similar fashion.

*This work was supported in part by the IISC-IBM joint project on High Performance Computing using Distributed Shared Memory under the Shared University Research Program.
2. Algorithm for the NSP

2.1. Preliminaries

Before presenting the algorithm we define some functions that are used by the algorithm.

1. **BDPRECOMP**: Performs prefix computation on a sequence $IED$ stored on a $p$-processor $BDM$ machine. For details refer Theorem A.1 in Appendix A.

2. **BLOCKMERGE**: Merges two sorted lists $L_1$ and $L_2$ each of length $t(n/p)$ elements, $t > 0$ an integer, such that $L_1$ is $IED$ stored on an array $BL_1$ in the processors $PR_i, PR_{i+1}, \ldots, PR_{i+(t-1)}$ and $L_2$ is $IED$ stored on an array $BL_2$ in the processors $PR_{i+a}, PR_{i+(a+1)}, \ldots, PR_{i+(2t-1)}$, for some integer $a > 0$, and outputs the merged sorted list $L$ of length $2t(n/p)$, $IED$ stored on an array $BL$ in the processors $PR_i, PR_{i+a}, \ldots, PR_{i+(2t-1)a}$. For details refer Theorem A.2 and the discussion preceding it in Appendix A.

3. **RANDOMROUTE**: A randomized function which routes the data stored in each of the processors to their respective destinations. The input to this function is a $[n] \times p$ array $A$ of $n$ elements initially stored one column per processor in a $p$-processor $BDM$ machine. Each element of $A$ consists of a pair $(i, data)$, where $i$ is the index of the processor to which the data has to be relocated. For details refer Theorem A.3 in Appendix A.

2.2. The Algorithm

The Sequential Algorithm.

**Definition 3** Let $A = (\Gamma, a_1, a_2, \ldots, a_n)$ be an array of elements from a totally ordered domain, where $\Gamma$ is a dummy element such that $\Gamma < a_i$, $1 \leq i \leq n$. Then,

1. $N_A := (a_{j_1}, a_{j_2}, \ldots, a_{j_k})$ such that $j_1 = n$, $a_{j_1+1}$ is the left match of $a_{j_1}$, $1 \leq i < k$, and $a_{j_k} = \Gamma$.

2. $M_A := (c_1, c_2, \ldots, c_r)$ is a subsequence of $A$ comprising of the elements which do not have a left match, listed in the same order as they appear in $A$.

3. $L_A$ is the list of elements having left matches along with their left matches.
We assume the standard stack operations Push(e, S), which pushes the element e onto a stack S, Top(S) which returns the topmost element of S, and Pop(S) which returns and removes the topmost element of S.

**Function SEQNSP(A) : (NA, MA, LA)**

**Input:** An array \( A = (\Gamma, a_1, a_2, \ldots, a_n) \) of elements from a totally ordered domain.

**Output:** \( N_A, M_A \) and \( L_A \) as defined in Definition 3.

**begin**
1. Let \( S \) be an empty stack; \( N_A := M_A := L_A := \emptyset \);
2. Push(\( \Gamma, S \));
3. for \( i = 1 \) to \( n \) do
4. while \( (a_i < \text{TOP}(S)) \) Pop(S); endwhile;
5. if (\( \text{TOP}(S) = \Gamma \)) then append \( (a_i) \) to \( M_A \);
6. else append \( (< a_i, \text{TOP}(S) >) \) to \( L_A \);
7. Push(\( a_i, S \));
8. endfor;
9. \( N_A := \) contents of \( S \) from top to bottom;
10. return(\( N_A, M_A, L_A \))
**end.**

**Theorem 1** Given an array \( A \) of \( n \) elements, the function SEQNSP(\( A \)) takes \( O(n) \) time.

**The Distributed Algorithm for DSM systems.**

We assume that the input array \( A \) is of the form \( (\Gamma, a_1, a_2, \ldots, a_n) \). W.l.o.g. we assume that \( n \) is a power of 2. We first present a divide-and-conquer algorithm for the NSP which we then parallelize: Split \( A \) into two halves \( A_1 = (\Gamma, a_1, a_2, \ldots, a_{k/2}) \) and \( A_2 = (\Gamma, a_{k/2+1}, a_2, \ldots, a_n) \) and solve the NSP for \( A_1 \) and \( A_2 \) separately. Let the solutions for \( A_1 \) and \( A_2 \) be \( (N_{A_1}, M_{A_1}, L_{A_1}) \) and \( (N_{A_2}, M_{A_2}, L_{A_2}) \), respectively. We will now see how to compute \( (N_A, M_A, L_A) \).

**Lemma 1** If an element \( e \in M_{A_2} \) has a left match \( l \), then \( l \in N_{A_1} \).

**Proof:** Obviously, \( l \in A_1 \). Let \( N_{A_1} = (a_{j_1}, a_{j_2}, \ldots, a_{j_k}) \). Suppose \( l \not\in N_{A_1} \), then \( l = a_r \) such that \( j_i > r > j_{i+1} \) for some \( 1 \leq i < k \). We obtain easily that \( a_r > a_j \) and \( j_i > r \). This implies that \( a_j \) is closer to \( e \) than \( l \) and less than \( e \), contradicting our assumption that \( l \) is the left match of \( e \).

We easily see that \( M_{A_2} \) and \( N_{A_1} = (a_{j_1}, a_{j_2}, \ldots, a_{j_k}) \) are sorted in decreasing order. Merge \( M_{A_2} \) with \( N_{A_1} \) to obtain \( T_A := (a_{j_1}, a_{j_2}, \ldots, a_{j_k}) \) sorted in decreasing order. Form an auxiliary array \( P[1..t] \) such that for every element \( a_{j_i} \in T_A \), \( P[i] := < a_{j_i}, a_{j_{i+1}} > \) if \( a_{j_i} \in N_{A_1} \) and \( P[i] := < 0, \Gamma > \) otherwise, \( 1 \leq i \leq t \). We then compute the suffix maxima \( PMAX \), on the first entries of the \( P \)-array elements. We easily obtain the following

**Lemma 2** Let \( T_A = (a_{j_1}, a_{j_2}, \ldots, a_{j_t}) \). For \( a_{j_i} \in M_{A_2} \), \( 1 \leq i \leq t \), we have

1. if \( PMAX[i] = < 0, \Gamma > \) then, \( a_{j_i} \) does not have a left match in \( A \).
2. if \( PMAX[i] = < f_r, a_{j_r} > \) then, \( a_{j_r} \) is the left match for \( a_{j_i} \) in \( A \).

Now, our method to compute \( N_A, M_A \) and \( L_A \) from \( N_{A_1}, M_{A_1}, L_{A_1}, N_{A_2}, M_{A_2}, \) and \( L_{A_2} \) is as follows: The list \( M_A \) is the list \( M_{A_1} \) appended to its tail the list of elements of \( M_{A_2} \) that satisfy condition 1 of Lemma 2, in the same order as they appear in \( M_{A_2} \). The elements of \( L_A \) are the elements of \( L_{A_1}, L_{A_2} \) and those elements of \( M_{A_2} \) that satisfy condition 2 of Lemma 2. The list \( N_A \) consists of \( N_{A_2} \) (without last element \( \Gamma \)) appended to its tail all elements in \( N_{A_1} \) that are less than every element in \( M_{A_2} \). Let \( e \) be the last, i.e. smallest, element of \( M_{A_2} \) and \( a_j < e \leq a_{j_{k+1}} \) for \( N_{A_1} = (a_{j_1}, a_{j_2}, \ldots, a_{j_k}) \); since \( a_{j_k} = \Gamma \) such an \( i \), \( 1 \leq i \leq k \) exists. Hence, \( N_{A_2} \) is the list \( N_{A_2} \) appended to its tail the list \( (a_{j_{k+1}}, \ldots, a_{j_k}) \). Definition 3 and Lemmas 1 and 2 imply the correctness of this method. Example 1 gives an illustration:

**Example 1** (Divide and Conquer NSP) Let \( A = (\Gamma, 7, 3, 2, 4, 6, 8, 1, 5) \). Hence, \( A_1 = (\Gamma, 7, 3, 2, 4) \) and \( A_2 = (\Gamma, 6, 8, 1, 5) \). We obtain \( M_{A_1} = (7, 3, 2) \), \( N_{A_1} = (4, 2, \Gamma) \), \( L_{A_1} = (4, 2, >) \), \( M_{A_2} = (6, 1) \), \( N_{A_2} = (5, 1, \Gamma) \), \( L_{A_2} = (8, 6, >, 5, 1, >) \), \( T_A = (6, 4, 2, 1, \Gamma) \), \( P = (6, <, 0, \Gamma, >, 4, 4, >, 3, 2, >, 0, 0, >, 0, >) \), and \( PMAX = (4, 4, >, 4, 4, >, 3, 2, >, 0, 0, >, 0, >) \). Our merging method then returns \( L_A = M_{A_1} + L_{A_2} + (< 6, 4 >) \); \( M_A = M_{A_1} + (1) = (7, 3, 2, 1) \); and \( N_A = (5, 1, \Gamma) \).

Now, the distributed algorithm works recursively, basically boiling down to the assumption that the n elements of the input A are IED stored on the array DA on the processors \( PR_0, PR_1, \ldots, PR_{p-1} \) of the p processors BDM machine. The BDM model permits to assume such initial placements (fact 2 on page 2).

W.l.o.g. we assume that \( p \) is a power of 2. We further assume that also \( L_A, M_A, N_A \) are IED stored on arrays \( DLDA_A, DMDA_A, DND_A \), respectively, on the processors \( PR_0, PR_1, \ldots, PR_{p-1} \). Each processor \( PR_i, 0 \leq i \leq p-1 \), solves the NSP for the elements stored within itself sequentially using the function SEQNSP presented before. The p individual solutions are merged using a method similar to the divide-and-conquer approach discussed above to get the final
solution. The function BLOCKNSP below gives the pseudocode for this recursive DSM algorithm. We assume that $DL_{DA,n,p}$ are global array and global variables, respectively, since this allows outputting the left match of an element $e$ at any level of recursion. At the end of every level of recursion $DL_{DA}$ is updated using the RANDOMROUTE function for all elements which found left matches during that level. All elements in the $DL_{DA}$ array are assumed to be initialized to $\Gamma$ at the beginning.

Function

**BLOCKNSP**$(DA,S,E) : (DN_{DA},DM_{DA})$

**Input:** The sequence $A$ of $(E - S + 1)n/p$ elements as defined above, $I\in D$ stored on the array $DA$ in the processors $PR_1, PR_{S+1}, \ldots, PR_E$, $E \geq S$.

**Output:** $DN_{DA}$ and $DM_{DA}$ for $DA$, such that every entry of $DN_{DA}$ and $DM_{DA}$ is a dummy or is of the form $< DA[j,i], j, i >$. The elements of $DN_{DA}$ and $DM_{DA}$ are assumed to be initialized to be dummy elements at the beginning.

**begin**

1. if $(S = E)$ then solve the NSP for the $n/p$ elements stored in the sequence $DA[j,S]$, $1 \leq j \leq n/p$, sequentially using the function SEQNSP; from the result form the arrays $DN_{DA}[j,S], DM_{DA}[j,S], DL_{DA}[j,S]$, $1 \leq j \leq n/p$;

   return($DN_{DA}, DM_{DA}$);

2. do in parallel /* Corresponds to computing $N_1, M_1, N_2, M_2, L_1, L_2$ in Example 1. */

   $(DN_{DA}^1, DM_{DA}^1) := BLOCKNSP(DA,S, E-\frac{S+1}{2});$

   $(DN_{DA}^2, DM_{DA}^2) := BLOCKNSP(DA, E-\frac{S+1}{2} + 1, E);$

3. $DT_{DA} := BLOCKMERGE(DM_{DA}^2, DN_{DA}^1, S, E-\frac{S+1}{2}, 1);$ /* Corresponds to computing $T_4$ in Example 1. From the definitions of $DM_{DA}$ and $DN_{DA}$ we see that every element of $DT_{DA}$ will be of the form $< DA[j,i], j, i >$. */

4. for each processor $PR_i, S \leq i \leq E$, do in parallel,

   for $j \leftarrow 1$ to $n/p$ do sequentially /* Let
   $DT_{DA}[j,i] := < e, f, g > */$

   if $(g < \frac{E-\frac{S+1}{2}}{2}) / * e \in DM_{DA}^1$ then

   $DP[j,i] := < 0, \Gamma >$

   else /* $e \in DN_{DA}^1$ */

   $DP[j,i] := < g * (n/p) + f, e > ;$

   /* Corresponds to computing the list $P$ in Example 1. */

5. $DPMAX := BDPRECOMP(DP, Max);$ /* Corresponds to computing the list $PMAX$ in Example 1. */

6. $DPMIN := BDPRECOMP(DP, Min);$ /* Will be useful in finding $DN_{DA}$. */

7. for each processor $PR_i, S \leq i \leq \frac{E-\frac{S+1}{2}}{2},$ do in parallel,

   for $j \leftarrow 1$ to $n/p$ do sequentially /* All elements of $DM_{DA}^1$ belong to $DM_{DA}$. */

   $DM_{DA}[j,i] := DM_{DA}^1[j,i];$

8. for each processor $PR_i, \frac{E-\frac{S+1}{2}}{2} + 1 \leq i \leq E,$ do in parallel,

   for $j \leftarrow 1$ to $n/p$ do sequentially /* All elements of $DN_{DA}^2$ belong to $DN_{DA}$. */

   $DN_{DA}[j,i] := DN_{DA}^2[j,i];$

9. for each processor $PR_i, S \leq i \leq E,$ do in parallel

   for $j \leftarrow 1$ to $n/p$ do sequentially /* Let
   $DT_{DA}[j,i] := < e, f, g > */$

   if $(g \leq \frac{E-\frac{S+1}{2}}{2})$ then /* $e \in DN_{DA}^1$ */

   $DPMIN[j,i] := < a, b > ;$

   if $(a \neq 0)$ then from the definition of $DP$ we can see that there does not exist any element of $DM_{DA}^2$ that is less than $e$ in $DT_{DA}$. From Example 1 and the discussion preceding it we see that $e \in DN_{DA}$ and hence $DN_{DA}[j,f] := < e, f, g >$. This has to be updated in processor $g$. Hence add $< g, < f, e >$ to an array $TEMP_{P_1}[1..n/p,i]$;

   else /* $e \in DM_{DA}^2$ */

   let $DPMAX[j,i] := < a, b > ;$

   if $(b = \Gamma)$ then from the definition of $DP$ we can see that $e$ has no left match. Hence $DM_{DA}[j,f] := < e, f, g >$. This has to be updated in processor $g$. Hence add $< g, < f, e >$ to an array $TEMP_{P_2}[1..n/p,i]$;
if \( b \neq \Gamma \) then from the definition of \( DP \) we can see that \( b \) is the left match of \( e \). Hence \( DL_{DA}[f, g] = e \). This has to be updated in processor \( g \). Hence add \(< g, < f, e > >\) to an array \( TEMP_{P3}[1..n/p, i] \);

/* The arrays \( TEMP_{P1}, TEMP_{P2} \) and \( TEMP_{P3} \) contain data to be routed. */

10. \((TEMP_{P1}[i], c_{1}) := RANDOMROUTE(TEMP_{P1});\)
    \((TEMP_{P2}[i], c_{2}) := RANDOMROUTE(TEMP_{P2});\)
    \((TEMP_{P3}[i], c_{3}) := RANDOMROUTE(TEMP_{P3});\)

/* The function RANDOMROUTE and Theorem A.3 can be applied to \( TEMP_{P1}, TEMP_{P2} \) and \( TEMP_{P3} \). RANDOMROUTE updates the arrays \( DN_{DA}[1..n/p, i], DM_{DA}[1..n/p, i] \) and \( DL_{DA}[1..n/p, i] \);

11. for each processor \( PR_{i}, S \leq i \leq E \), do in parallel
    Scan the arrays \( TEMP_{P1}[1..c_{1}(\frac{n}{p}), i], TEMP_{P2}[1..c_{2}(\frac{n}{p}), i], TEMP_{P3}[1..c_{3}(\frac{n}{p}), i] \) and update the arrays \( DN_{DA}[1..n/p, i], DM_{DA}[1..n/p, i] \) and \( DL_{DA}[1..n/p, i] \);

12. return \((DN_{DA}, DM_{DA})\)

/* Note that, if we scan the arrays \( DN_{DA}[j, i] \) and \( DM_{DA}[j, i] \) in order from \( j = 1 \) to \( j = n/p \) and \( i = 0 \) to \( i = p-1 \) leaving the dummies we get \( N_{A} \) and \( M_{A} \) as in Example 1, respectively, in increasing order. */

At the end of execution of the function BLOCKNSP, the following facts about \( e := DA[j, i] \) are satisfied:

1. If there exists a left match \( l \) for \( e \) then,
   \( DL_{DA}[j, i] = e \) and \( DM_{DA}[j, i] = \text{dummy} \).
2. If there is no left match for \( e \) then, \( DL_{DA}[j, i] = \Gamma \) and \( DM_{DA}[j, i] = < e, j, i > \).
3. If \( e \in DN_{DA} \) then, \( DN_{DA}[j, i] = < e, j, i > \).

**Time Complexity.**

From Theorem 1 and the Function BLOCKNSP we see that steps 1, 4, 7, 8, and 9 take \( O(n/p) \) time. Theorem A.2 implies that step 3 takes \( O\left(\frac{n \log_{p}(E - \frac{S+1}{2})}{p}\right) \) computation time and \( O\left((\tau + \sigma m \left[\frac{n}{pm}\right]) \log_{p}(E - \frac{S+1}{2})\right) \) communication time. Theorem A.1 implies that steps 5 and 6 take \( 4r \frac{\tau \log_{p}(E - \frac{S+1}{2})}{\log_{p}(E - \frac{S+1}{2})} \) + \( \tau + \sigma m \) communication time and \( O\left(\frac{n \log_{p}(E - \frac{S+1}{2})}{p} + \frac{\tau \log_{p}(E - \frac{S+1}{2})}{\sigma m \log_{p}(E - \frac{S+1}{2})}\right) \) computation time. For each execution of the RANDOMROUTE function in step 10 at most \( n/p \) elements are destined per processor. Hence \( \alpha \leq 1 \) in Theorem A.3. Substituting for \( \alpha \) in Theorem A.3 we see that step 10 takes \( 2(\tau + c_{E} \left[\frac{n}{p}\right]) \) communication time and \( O(c_{E} \left[\frac{n}{p}\right]) \) computation time with high probability, where \( c \) is a constant. Theorem A.3 also implies that \( c_{1}, c_{2} \) and \( c_{3} \) in step 10 can be output in constant time. Hence, step 11 takes \( O(n/p) \) time. Let \( t := E - S + 1 \) and \( T_{comp}(t) \) be the computation time taken by BLOCKNSP \((DA, S, E)\). From Theorem 1 we see that \( T_{comp}(1) = O(n/p) \). From function BLOCKNSP we see that,

\[
T_{comp}(t) = T_{comp}(\frac{t}{2}) + O\left(\frac{n \log_{p} t}{p} + \frac{\tau \log_{p} t}{\sigma m \log_{p}(\frac{t}{\sigma m} + 1)}\right)
\]

\[
= O\left(\frac{n \log_{p} t}{p} + \frac{\tau \log_{p} t}{\sigma m \log_{p}(\frac{t}{\sigma m} + 1)}\right)
\]

Let \( T_{comm}(t) \) be the communication time taken by BLOCKNSP \((DA, S, E)\). From Theorem 1 we see that \( T_{comm}(1) = 0 \). From function BLOCKNSP we see that,

\[
T_{comm}(t) = T_{comm}(\frac{t}{2}) + O\left((\tau + \sigma m \left[\frac{n}{pm}\right]) \log_{p} t\right)
\]

\[
= O\left((\tau + \sigma m \left[\frac{n}{pm}\right]) \log_{p} t\right)
\]

Given the input array \( DA \), the function is called as BLOCKNSP \((DA, 0, p-1)\). This and the above discussion imply the following theorem.

**Theorem 2** The \( NSP \) can be solved in 
\( O\left((\tau + \sigma m \left[\frac{n}{pm}\right]) \log_{p} \right) \) computation time and 
\( O\left(\frac{n \log_{p} p}{p} + \frac{\tau \log_{p} p}{\sigma m \log_{p}(\frac{p}{\sigma m} + 1)}\right) \) computation time on a DSM system using the BDM model of computation.

**3. Applications.**

Our BDM-based algorithm for the \( NSP \) gives BDM-based algorithms for the following problems, that are mapped onto the \( NSP \) in \([1]\):

1. **Triangulating monotone polygons** (a monotone polygon is one that can be split into two monotone polygonal chains such that the vertices of the chains are increasing (or decreasing) by the \( x \)-coordinate).
2. **Reconstruction of binary trees from their traversals** (from inorder and preorder traversals).
3. Parentheses matching (find the level of nesting for each parenthesis in a legal sequence of parenthesis, and also find for each parenthesis its left mate).

4. Conclusion

In this paper we presented a simple and efficient algorithm for the nearest smaller problem (NSP), which, to the best of our knowledge is the first of its kind for DSM systems. Since the NSP is fundamental in many problems, a solution for it on DSM systems implies DSM-based solutions for a variety of problems in diverse areas as discussed in this paper.

A. Appendix

Function BDPRECOMP(A, ) : (A').
Input: A sequence of ordered pairs < a1, data1 >, < a2, data2 >, ..., < ar, data r >, IED stored on the array A on a p-processor BDM machine, datai is the data (if any) associated with ai, 1 ≤ i ≤ r. is a binary associative operator ∈ {+, Min, Max, ...}.
Output: A sequence of ordered pairs < a'1, data'1 >, < a'2, data'2 >, ..., < a'r, data'r >, IED stored on the array A' on the p-processor BDM machine, where, a'i = i=1 a k and datai' is the data associated with a'i (if any).

From Theorem 9 of [4] we infer the following theorem.

Theorem A.1 ([4]) Given a sequence (a1, a2, ..., ar) of numbers IED stored on a p-processor BDM, we can compute the prefix sums psi = i=1 ai, 1 ≤ i ≤ r, in 4τ[log2(p) + log2(p)] + τ + σm communication time and O(n + r log2(p) + σm log2(p)) computation time. This complexity holds for prefix maxima, prefix minima and similar associative operators.

Theorem A.2 ([3]) Function BDMERGE(BL1, BL2, i, t) takes O(n log2 t) computation time and O((τ + σm[log2 t])) communication time.

Function RANDOMROUTE(A) : (A', c)
Input: Input array A[1 : [n/p], 0 : p - 1] IED stored on a p-processor BDM machine, such that each element of A consists of a packet (i, datai) of constant size, where i is the index of the processor to which datai has to be routed. c is a constant such that no processor is the destination of more than c[log2 n] elements on the whole.

Output: Output array A'[1 : [n/p], 0 : p - 1] holding the routed data IED stored on a p-processor BDM machine, such that all the data with the processor PRi, 0 ≤ i ≤ p - 1, as the destination will be available in one of the locations A'[i, j], 1 ≤ j ≤ c[log2 n], in the processor PRj, where c is larger than max{1 + [1/2], α + √2}, The function stores a copy of c in every processor PRi, 0 ≤ i ≤ p - 1.
The function is implemented using the randomized routing algorithm suggested in [2].

Theorem A.3 The function RANDOMROUTE(A) completes within 2τ + c[log2 n] communication time and O(c[log2 n]) computation time with high probability, where c is larger than max{1 + [1/2], α + √2}, p2 < 8n and α is such that every processor is a destination for at most c[log2 n] messages. □

References