Efficient Algorithms for Prefix and General Prefix Computations on Distributed Shared Memory Systems with Applications *

V. Kamakoti N. Balakrishnan

Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore - 560 012, India
{kama,balki}@serc.iisc.ernet.in

Abstract

The paper presents efficient scalable algorithms for performing Prefix (PC) and General Prefix (GPC) Computations on a Distributed Shared Memory (DSM) system with applications.

1 Introduction

Prefix (*PC*) and *General Prefix* (*GPC*) Computations are generic techniques that can be used to design sequential and parallel algorithms for a number of problems from diverse areas [1, 5, 6]. In [6] an $O(\log n)$ time, $O(n)$ processor, *CREW PRAM* (shared memory) parallel algorithm for the *GPC* is presented. This algorithm implies an $O(n \log n)$ time sequential algorithm.

Based on 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. A software abstraction in which a distributed memory system can be viewed as a system with 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.* The *BDM* model of computing, from a users perspective, offers the advantage of ease of programming of shared memory systems while from a systems perspective, provides the advantage of scalability akin to message passing systems.

PC and *GPC* are generic techniques for algorithm design. Therefore, a solution for them implies a solution for a variety of problems in diverse areas that include Computational Geometry, Graph Theory, Sorting etc. Some of the routines are recursive and were successfully implemented on the SP2 which is a Distributed Memory Machine. This gives some pointers for automatically parallelizing recursive programs. The paper also discusses the suitability of the *BDM* model [3] for the IBM's SP2. This implies that the model can serve as a tool for the programmer to design and analyse the algorithm, by deriving analytical expressions for the time/space complexity before actually implementing the same. In this regard, our paper can also serve as a case study for the users. We now formally define *PC* and *GPC* with an example shown in Figure 1.

m	1	2	3	4	5	6
f	1.2	1.6	1.3	1.2	1.7	1.4
C_m	1.2	1.6	1.6	1.6	1.7	1.7
Y	1	4	2	3	4.5	3.5
D_m	ϕ	1.2	1.2	1.3	1.6	1.3

$n = 6$, $* =$ max, $\rho = <$, $\phi =$ not defined.

Figure 1: Example for Prefix Maxima and General Prefix Maxima

Definition 1 (Prefix Computation (*PC*)) *Let $(f(1), f(2), \cdots, f(n))$ be a given sequence of elements. The problem is to compute the sequence of prefixes $C_m = f(1) * f(2) * \cdots * f(m)$, $1 \leq m \leq n$, where, $*$ is an arbitrary binary associative operator defined on the f-elements.*

Definition 2 (*GPC* [6]) *Let $(f(1), f(2), \cdots, f(n))$ and $(y(1), y(2), \cdots, y(n))$ be two given sequences of elements. Let $Y = \{y(1), ..., y(m)\}$. Let ρ be a binary relation such that for $y_i, y_j \in Y$, $i \neq j$, either $(y_i \ \rho \ y_j)$ or $(y_j \ \rho \ y_i)$. The problem is to compute the sequence of general prefixes $D_m = f(j_1) * f(j_2) * \cdots * f(j_k)$, $1 \leq m \leq n$, where, $*$ is an arbitrary binary associative operator defined on the f-elements and j_i's are indices such that*

(a). $j_i < j_{i+1}$, $1 \leq i < k$;
(b). $j_i < m$, $1 \leq i \leq k$; and,
(c). $y(j_i) \ \rho \ y(m)$, $1 \leq i \leq k$.

It is easy to see that *PC* is a special instance of *GPC*. An efficient algorithm for *PC* is reported in [4].

*This work was supported in part by the IISC-TISL/IBM joint project (No. TISL/SERC/NB/02) on *High Performance Computing Using Distributed Shared Memory* under the Shared University Research Program

0-8186-8227-2/97 $10.00 © 1997 IEEE

1.1 Preliminaries

The *BDM* model [3] is defined in terms of four parameters: *number p of processors*, *maximum initial latency time τ* taken for a processor to receive the packet it requested from some other processor, *time σ taken to inject a word into or receive a word from the network*, and *number m of consecutive words sent during each transfer*. The processors are connected to a common communication network. Data are communicated between processors via point-to-point messages in blocks of m consecutive words rather than a single word. This is done keeping in mind the *spatial locality* of programs in execution. Let PR_i, $0 \leq i \leq p-1$ denote the i^{th} processor in the *BDM* system. Any processor can communicate with any other processor, but the time for communication depends upon the latency and bandwidth of the network, as described in the following facts about the *BDM* given in [3]:

1. No processor can send or receive more than one *packet* (a block of m consecutive words) at a time.

2. The model allows the *initial placement* of input data in the local memories of the processors and the memory latency hiding technique of *pipelined prefetching.*

3. If π is any permutation on p elements, then, a remote memory request for b words issued by every processor PR_i and destined for processor $PR_{\pi(i)}$ can be completed in $\tau + m\sigma\lceil\frac{b}{m}\rceil$ time for *all* processors PR_i, $0 \leq i \leq p-1$, simultaneously. k remote access requests issued by k distinct processors and destined to the same processor will require $k(\tau + m\sigma)$ time to be completed, and the requests will be served in arbitrary order. k prefetch read operations issued by a processor can be completed in $\tau + km\sigma$ time, using *pipelined prefetching.* k prefetch read operations of k blocks of $\lceil\frac{n}{p}\rceil$ words each, can be completed in $\tau + \sigma km\lceil\frac{n}{pm}\rceil$ time.

4. There are two time-complexity measures for a parallel algorithm on the *BDM* model; the computation time T_{comp}, and the communication time T_{comm}. The measure T_{comp} refers to the maximum of the local computations performed on any processor as measured in the model of computation supported by it. The measure T_{comm} refers to the total amount of communication time spent by the overall algorithm in accessing remote data.

Henceforth, without loss of generality we assume that n is a multiple of p and p is a power of two.

Definition 3 ((IED) storage) *A sequence* $F = (f(1), f(2), ..., f(n))$ *of n elements is said to be Inorder Equally Distributed IED stored on a two-dimensional array* $B[1..\frac{n}{t} : 0..t-1]$ *in some* $t \leq p$ *processors* $PR_{j_0}, PR_{j_1}, ..., PR_{j_{t-1}}$ *of a BDM machine if and only if,*

> $B[j,i] = f(i*(n/t)+j)$, *for* $0 \leq i \leq t-1$, $1 \leq j \leq n/t$
>
> $(B[1,i], B[2,i], ..., B[\frac{n}{t}, i])$ *are stored in processor* PR_{j_i} *in this order,* $0 \leq i \leq t-1$.

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

1. **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 $\frac{n}{p} \times p$ 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 $(data, i)$, where i is the index of the processor to which the data has to be relocated. For details refer Theorem A.1 in Appendix *A*.

2. **BDPRECOMP**: Performs prefix computation on a sequence *IED* stored on a p-processor *BDM* machine. For details refer Theorem A.2 in Appendix *A*.

2 Merging Sorted Lists on a *BDM* machine

We will first present an overview of the function which merges two given sorted lists on a *BDM* machine and then, present its implementation. The function is a modification of Batcher's odd-even merge [2]. Without loss of generality we assume $p = 2^b$, for some integer $b > 0$. The two given sorted lists L_1 and L_2 of $\frac{n}{2}$ elements each are such that L_1 is *IED* stored in order in the processors $(PR_0, PR_1, ..., PR_{\frac{p}{2}-1})$ and L_2 is *IED* stored in order in the processors $(PR_{\frac{p}{2}}, PR_{\frac{p}{2}+1}, ..., PR_{p-1})$, $\lceil\frac{n}{p}\rceil$ elements per processor. We call the data stored within a single processor as a *block.* As in *odd-even merge* we do the following steps 1 and 2 in parallel:

1. Recursively merge the blocks stored in processors occupying even numbered positions/subscripts. In other words, the sorted sublist of L_1 stored in $(PR_0, PR_2, ..., PR_{\frac{p}{2}-2})$ and the sorted sublist of L_2 stored in $(PR_{\frac{p}{2}}, PR_{\frac{p}{2}+2}, ..., PR_{p-2})$ are merged and the merged list is stored in $(PR_0, PR_2, ..., PR_{\frac{p}{2}-2}, PR_{\frac{p}{2}}, PR_{\frac{p}{2}+2}, ..., PR_{p-2})$.

2. Recursively merge the blocks stored in processors occupying odd numbered positions/subscripts. In other words, the sorted sublist of L_1 stored in $(PR_1, PR_3, ..., PR_{\frac{p}{2}-1})$ and the sorted sublist of L_2 stored in $(PR_{\frac{p}{2}+1}, PR_{\frac{p}{2}+3}, ..., PR_{p-1})$ are merged and the merged list is stored in $(PR_1, PR_3, ..., PR_{\frac{p}{2}-1}, PR_{\frac{p}{2}+1}, PR_{\frac{p}{2}+3}, ..., PR_{p-1})$.

3. For all k, $0 \leq k \leq p-1$, **do in parallel**

   if k is odd and $k \neq p-1$, then, merge the block stored in PR_k with the block stored in PR_{k+1}.

4. The final sorted list is stored in order in $(PR_0, PR_1, ..., PR_{p-1})$.

The procedure is similar to the odd-even merge [2] except that we treat every element of the latter as a block of sorted elements. The following function *BlockMerge* gives the recursive *DSM* implementation of the above method.

Function $BlockMerge(BL_1, BL_2, i, t, a) : (BL)$
Input: Two sorted lists L_1 and L_2 each of length $t(n/p)$ elements such that L_1 is *IED* stored on an array BL_1 in the processors $PR_i, PR_{i+a}, ..., PR_{i+(t-1)a}$ and L_2 is *IED* stored on an array BL_2 in the processors $PR_{i+ta}, PR_{i+(t+1)a}, ..., PR_{i+(2t-1)a}$.
Output: The sorted list $L = Merge(L_1, L_2)$, of length $2t(n/p)$ *IED* stored on an array BL in the processors $PR_i, PR_{i+a}, ..., PR_{i+(2t-1)a}$

```
Begin /* W.l.g we assume that $t$ is a power of two */
1.  If ($t = 1$) then /* only two blocks to be merged */
      $PR_i$ and $PR_{i+a}$ hold the sorted lists $BL_1$ and $BL_2$
      of $\lceil\frac{n}{p}\rceil$ elements each respectively. $PR_i$ reads the list
      $BL_2$ stored in $PR_{i+a}$ and sequentially merges it with
      $BL_1$ and stores one half of the merged list in itself,
      in the array $BL(j,i)$, $1 \le j \le n/p$ and the other half
      in $PR_{i+a}$ in the array $BL(j, i+a)$, $1 \le j \le n/p$.
      return($BL$); exit
Do Steps 2 and 3 in Parallel /* $t > 1$ */
2.  $BL = BlockMerge(BL_1, BL_2, i, \frac{t}{2}, 2a)$
/* In step 2 the part of the lists $BL_1$ and $BL_2$ stored in
      processors, $PR_{i+ka}$, for even $k \ge 0$ in the sequence
      $(PR_i, PR_{i+a}, \cdots, PR_{i+(2t-1)a})$ are merged */
3.  $BL = BlockMerge(BL_1, BL_2, i+a, \frac{t}{2}, 2a)$
/* In step 3 the part of the lists $BL_1$ and $BL_2$ stored in
      processors, $PR_{i+ka}$, for odd $k \ge 1$ in the sequence
      $(PR_i, PR_{i+a}, \cdots, PR_{i+(2t-1)a})$ are merged */
4.  For all Processors $PR_{i+ka}$, such that $1 \le k \le t-3$
            and $k$ is an odd number do in parallel
      $PR_{i+ka}$ reads the sorted list $BL(j, i+(k+1)a)$,
      $1 \le j \le n/p$ from $PR_{i+(k+1)a}$ and merges sequentially
      the same with the list $BL(j, i+ka)$, $1 \le j \le n/p$
      stored in it. Then, it stores one half of the merged list
      in itself, in the array $BL(j, i+ka)$, $1 \le j \le n/p$ and
      the other half in $PR_{i+(k+1)a}$ in the array
      $BL(j, i+(k+1)a)$, $1 \le j \le n/p$.
/* In step 4 all the processors, $PR_{i+ka}$, for odd $k \ge 1$, in
      the sequence $(PR_i, PR_{i+a}, \cdots, PR_{i+(2t-1)a})$ except
      the last processor $PR_{i+(2t-1)a}$, merges the $BL$ array
      stored in it with the $BL$ array of the immediately
      succeeding processor $PR_{i+(k+1)a}$. */
5.  return($BL$)
End.
```

From the assumptions of the *BDM* model we see that step 4 of the above function takes $\tau + \sigma m\lceil\frac{n}{pm}\rceil$ communication time. Step 4 takes $O(\frac{n}{p})$ computation time. Hence for the whole procedure,

$T_{comm}(t) = T_{comm}(\frac{t}{2}) + \tau + \sigma m\lceil\frac{n}{pm}\rceil$,
$T_{comm}(1) = \tau + \sigma m\lceil\frac{n}{pm}\rceil$ (from Step 1).
$T_{comp}(t) = T_{comp}(\frac{t}{2}) + O(\lceil\frac{n}{p}\rceil)$,
$T_{comp}(1) = O(\lceil\frac{n}{p}\rceil)$ (from Step 1).

Theorem 1 *Function* $BlockMerge(BL_1, BL_2, i, t, a)$ *takes* $O(\frac{n\log_2 t}{p})$ *computation time and* $O((\tau + \sigma m\lceil\frac{n}{pm}\rceil)\log_2 t)$ *communication time.*

Corollary 1 *Two lists each of size n can be merged using p processors on a distributed memory system in* $O(\frac{n\log_2 p}{p})$ *computation time and* $O((\tau + \sigma m\lceil\frac{n}{pm}\rceil)\log_2 p)$ *communication time.*

3 The Ranking Problem

We will now define a subroutine *RANKP* to solve the *ranking problem. RANKP* will be used by the main algorithm. First we define the following notations. Let $L = (l_1, l_2, ..., l_{t(n/p)})$ and $LD = ((y_1, d_1), (y_2, d_2), ..., (y_{t(n/p)}, d_{t(n/p)}))$ be two sequences *IED* stored on the arrays BL and BLD respectively in the processors $PR_c, PR_{c+1}, ..., PR_{c+(t-1)}$, $((n/p)$ elements per processor). Similarly, let $R = (r_1, r_2, ..., r_{t(n/p)})$ and $RD = ((y'_1, d'_1), (y'_2, d'_2), ..., (y'_{t(n/p)}, d'_{t(n/p)}))$ be two sequences *IED* stored on the arrays BR and BRD respectively in the processors $PR_{c+t}, PR_{c+t+1}, ..., PR_{c+(2t-1)}$. From Definition 3 we see that, $BLD[j,i] = (y_{i*(n/p)+j}, d_{i*(n/p)+j})$, $1 \le j \le n/p$, $0 \le i \le t-1$, and is stored in the processor PR_{c+i}. Let $d_{i*(n/p)+j} = (j, c+i, dat)$, where *dat* denotes some data of constant size. Similarly $BRD[j,i] = (y'_{i*(n/p)+j}, d'_{i*(n/p)+j})$, $1 \le j \le n/p$, $0 \le i \le t-1$, and is stored in the processor PR_{c+t+i}. Let $d'_{i*(n/p)+j} = (j, c+t+i, dat)$, where *dat* denotes some data of constant size. Let SL be the sorted sequence of LD, sorted on the y values. SL is *IED* stored on the arrays BSL in the processors $PR_c, PR_{c+1}, ..., PR_{c+(t-1)}$. Let SR be the sorted sequence of RD, sorted on the y' values. SR is *IED* stored on the arrays BSR in the processors $PR_{c+t}, PR_{c+t+1}, ..., PR_{c+(2t-1)}$.

Definition 4 (Ranking Problem) *The problem is to merge the two sorted lists SL and SR (on the y and y' values) into a single sorted list PL and do the following:*

1. *for every element* $PL(j)$, $1 \le j \le 2t(n/p)$, *if* $PL(j)(= (y_i, d_i)) \in LD$, *then find the two consecutive elements* (y'_j, d'_j) *and* (y'_k, d'_k) *in SR such that* y_i *lies inbetween* y'_j *and* y'_k. *Let* $f_i = (d'_j, d'_k)$*; and,*

2. *for every element* $PL(j)$, $1 \le j \le 2t(n/p)$, *if* $PL(j)(= (y'_i, d'_i)) \in RD$, *then find the two consecutive elements* (y_j, d_j) *and* (y_k, d_k) *in SL such that* y'_i *lies inbetween* y_j *and* y_k. *Let* $f_i = (d_j, d_k)$.

Let $F = (f_1, f_2, ..., f_{2t(n/p)})$ and PL be IED stored on the arrays TF and RPL in the processors $PR_c, PR_{c+1}, ..., PR_{c+(2t-1)}$.

```
Function $RANKP(BSL, BSR, c, t) : (RPL, TF)$
Begin
  1. $RPL = BlockMerge(BSL, BLR, c, t, 1)$ (merging on the $y$ and $y'$ values).
  2. For each processor $PR_{c+i}$, $0 \leq i \leq 2t-1$ do in parallel
     For $1 \leq j \leq n/p$
       • Let $RPL[j, i] = (u, v)$. From entry $v$ we can find out whether the element belongs to $BSL$ or $BSR$.
       • If the element $RPL[j, i] (= (y_k, d_k))$ belongs to $BSL$ then, set $TEMP[j, i] = (i * (n/p) + j, d_k)$, else $TEMP[j, i] = (0, 0)$
  3. Do a prefix maxima on the $TEMP$ array using the function $BDPRECOMP$ (refer Theorem A.2 of Appendix A). Let the routine return the array $TEMP'$.
  4. For each processor $PR_{c+i}$, $0 \leq i \leq 2t-1$ do in parallel
     For $1 \leq j \leq n/p$
       • Let $RPL[j, i] = (u, v)$. From entry $v$ we can find out whether the element belongs to $BSL$ or $BSR$. If the element $RPL[j, i] (= (y'_k, d'_k))$ belongs to $BSR$ then, let $TEMP'[j, i] = (u', v')$. It is easy to see that $v'$ is the first entry of the tuple $f'_k$.

     /* In a similar fashion we can calculate the second entry of the tuple $f'_k$ for the elements in $BSR$. We can also calculate for all elements $RPL[j, i]$ belonging to $BSL$, the corresponding $f$-tuple in a similar fashion. Hence we can compute the $TF$ array.*/
End.
```

Note that steps 2 and 4 take $O(n/p)$ computation time. Theorems 1 and A.2 imply the following theorem.

Theorem 2 *Function RANKP(BSL,BSR,c,t) takes* $O(\frac{n \log_2 t}{p} + \frac{\tau \log_2 t}{\sigma m \log_2(\frac{\tau}{\sigma m}+1)})$ *computation time and* $O((\tau + \sigma m \lceil \frac{n}{pm} \rceil) \log_2 t)$ *communication time.*

4 Algorithm for the *GPC* on the *BDM* model

The overview of the algorithm is as follows. The algorithm is a recursive one, which basically boils down to each processor PR_i, $0 \leq i \leq p-1$, performing the GPC for the elements stored within itself sequentially, using the algorithm presented in [6] and then, merging the p individual solutions to get the final solution.

4.1 Sequential Algorithm for the GPC problem

In this section we will briefly describe the algorithm for the GPC, presented in [6]. It is a *Divide and Conquer* recursive algorithm. It takes as an input two sequences $SF = (sf(1), sf(2), ..., sf(k))$ and $SY = (sy(1), sy(2), ..., sy(k))$ and returns as an output, four sequences OSY, SD, SP, SE. Note that, SF corresponds to the f sequence and SY corresponds to the y sequence of Definition 2. Before defining these arrays we introduce the following notion.

Definition 5 *Given the input sequences SF and SY, two indices i and j, i is to the* **left** *of j if and only if $i < j$. i is* **below** *j if and only if $sy(i)$ ρ $sy(j)$.*

In this section $*$ and ρ have the same meanings as in Definition 2. OSY is the sorted list of the elements in SY. $SD(i)$ is the general prefix product of all the elements in SF, whose indices are to the left of i and below i, $1 \leq i \leq k$. In other words, $SD(i) = sf(j_1) * sf(j_2) * \cdots sf(j_t)$, $1 \leq i \leq k$ where, $\{j_1, j_2, \cdots, j_t\}$ is the set of all indices j such that $j < i$ and $sy(j)$ ρ $sy(i)$ and $j_r < j_{r+1}, 1 \leq r < t$. Similarly, $SP(i) = sf(j_1) * sf(j_2) * \cdots sf(j_t)$, $1 \leq i \leq k$ where, $\{j_1, j_2, \cdots, j_q\}$ is the set of all indices j such that $sy(j)$ ρ $sy(i)$ (indices below i) and $SE(i) = sf(j_1) * sf(j_2) * \cdots sf(j_t)$, $1 \leq i \leq k$ where, $\{j_1, j_2, \cdots, j_q\}$ is the set of all indices j such that $(sy(j)$ ρ $sy(i))$ or $(sy(j) = sy(i))$ and $j_r < j_{r+1}, 1 \leq r < t$ (indices below or equal to i). We define the following function:
Function $SEQGPC(SF, SY, k) : (OSY, SP, SD, SE)$
The function takes as input two arrays SF and SY of length k and outputs four arrays, OSY, SP, SD and SE, each of length k.

At every stage of the recursion, each of the input sequences (refer to Definition 2) $SF = (sf(1), sf(2), ..., sf(k))$ and $SY = (sy(1), sy(2), ..., sy(k))$ are divided into two halves. Let $LSF = (sf(1), sf(2), ..., sf(\frac{k}{2}))$ and $RSF = (sf(\frac{k}{2}+1), ..., sf(k))$. Similarly, let $LSY = (sy(1), sy(2), ..., sy(\frac{k}{2}))$ and $RSY = (sy(\frac{k}{2}+1), ..., sy(k))$. The two subproblems, namely, the *left* subproblem with input sequences LSF and LSY, and the *right* subproblem with input sequences RSF and RSY are solved recursively using function $SEQGPC$. The solution for the *left* subproblem are four arrays each of length $\frac{k}{2}$, denoted by $LOSY, LSD, LSP, LSE$ in order. Similarly, the solution for the *right* subproblem are four arrays each of length $\frac{k}{2}$, denoted by $ROSY, RSD, RSP, RSE$ in order.

The next step is to merge these two solutions to form the four arrays OSY, SD, SP, SE for the problem with input sequences SF and SY. Clearly OSY can be obtained by merging the two sorted lists $LOSY$ and $ROSY$. For every element $sy(i)$ in LSY, i.e., $1 \leq i \leq \frac{k}{2}$, we find the indices $LPOS(i)$ and $RPOS(i)$ of those two elements in RSY, such that $sy(i)$ is in between $sy(LPOS(i))$ and $sy(RPOS(i))$ in OSY and no other element of RSY is in between $sy(LPOS(i))$ and $sy(RPOS(i))$ in OSY. Similarly, for every element $sy(i)$ in RSY, i.e., $\frac{k}{2}+1 \leq i \leq k$, we find the indices $LPOS(i)$ and $RPOS(i)$ of those two elements in LSY, such that $sy(i)$ is in between $sy(LPOS(i))$ and $sy(RPOS(i))$ in OSY and no other element of LSY is in between $sy(LPOS(i))$ and $sy(RPOS(i))$ in OSY. It is straightforward to see that, $LPOS(i) =$ rank of $sy(i)$ in OSY − rank of $sy(i)$ in $LOSY$, for $sy(i) \in LSY$, i.e., $1 \leq i \leq \frac{k}{2}$. Similarly, $LPOS(i)$ = rank of $sy(i)$ in OSY − rank of $sy(i)$ in $ROSY$, for $sy(i) \in RSY$, i.e., $\frac{k}{2} + 1 \leq i \leq k$. $RPOS(i) = LPOS(i) + 1$, $1 \leq i \leq k$. Now we present the following lemma.

Lemma 1 (Merge Lemma [6])
1. If $1 \leq i \leq \frac{k}{2}$, then,
a. $SD(i) = LSD(i)$
b. $SP(i) = LSP(i) * RSP(LPOS(i))$,
  if $sy(LPOS(i)) = sy(i)$
c. $SP(i) = LSP(i) * RSE(LPOS(i))$,
  if $sy(LPOS(i))\ \rho\ sy(i)$
d. $SE(i) = LSE(i) * RSE(LPOS(i))$,
  if $sy(LPOS(i)) = sy(i)$
e. $SE(i) = LSE(i) * RSP(RPOS(i))$,
  if $(sy(LPOS(i))\ \rho\ sy(i)) \wedge (sy(i)\ \rho\ sy(RPOS(i)))$
f. $SE(i) = LSE(i) * RSE(RPOS(i))$,
  if $(sy(LPOS(i))\ \rho\ sy(i)) \wedge (sy(i) = sy(RPOS(i)))$
2. If $\frac{k}{2} + 1 \leq i \leq k$, then,
a. $SD(i) = LSP(LPOS(i)) * RSD(i)$,
  if $sy(LPOS(i)) = sy(i)$
b. $SD(i) = LSE(LPOS(i)) * RSD(i)$,
  if $sy(LPOS(i))\ \rho\ sy(i)$
c. $SP(i) = LSP(LPOS(i)) * RSP(i)$,
  if $sy(LPOS(i)) = sy(i)$
d. $SP(i) = LSE(LPOS(i)) * RSP(i)$,
  if $sy(LPOS(i))\ \rho\ sy(i)$
e. $SE(i) = LSE(LPOS(i)) * RSE(i)$,
  if $sy(LPOS(i)) = sy(i)$
f. $SE(i) = LSP(RPOS(i)) * RSE(i)$,
  if $(sy(LPOS(i))\ \rho\ sy(i)) \wedge (sy(i)\ \rho\ sy(RPOS(i)))$
g. $SE(i) = LSE(RPOS(i)) * RSE(i)$,
  if $(sy(LPOS(i))\ \rho\ sy(i)) \wedge (sy(i) = sy(RPOS(i)))$

Proof: We will prove Case 2a. For a given index i such that $\frac{k}{2}+1 \leq i \leq k$, let $\{j_1, j_2, \cdots, j_t\}$ is the set of all indices j such that $1 \leq j \leq \frac{k}{2}$ and $sy(j)\ \rho\ sy(i)$ and $j_r < j_{r+1}, 1 \leq r < t$. If $sy(LPOS(i)) = sy(i)$, then, it is straightforward to see that $LSP(LPOS(i)) = f(j_1) * f(j_2) * \cdots * f(j_t)$. This and definition of $RSD(i)$ imply $SD(i) = LSP(LPOS(i)) * RSD(i)$, if $sy(LPOS(i)) = sy(i)$. The proof for the other cases is similar. □

From the above discussion and Lemma 1 we see that the merge step can be implemented in $O(k)$ time. Hence the whole algorithm takes $O(k \log k)$ time.

Theorem 3 *Function* SEQGPC(SF, SY, k) *takes* $O(k \log k)$ *time.*

4.2 The Main Algorithm

Let us assume the two input sequences $F = (f(1), f(2), ..., f(t(n/p)))$ and $Y = (y(1), y(2), ..., y(t(n/p))))$ be IED stored on the arrays SF and SY respectively in the processors $PR_i, PR_{i+1}, ..., PR_{i+t-1}$. We also assume that the sequences OSY, SP, SD, SE, $LPOS$, $RPOS$ (as defined earlier are IED stored on the arrays $SOSY$, SSP, SSD ,SSE, $SLPOS$, $SRPOS$ respectively. The following recursive function solves the GPC problem for above input sequences SF and SY.

Function $BlockGPC(SF, SY, i, t) : (DATA)$
Input: Two input sequences $F = (sf(1), ..., sf(t\lceil \frac{n}{p} \rceil)$ and $Y = (sy(1), ..., sy(t\lceil \frac{n}{p} \rceil)$, each IED stored on the arrays SF and SY respectively (i.e., stored in the processors $PR_i, PR_{i+1}, ..., PR_{i+(t-1)})$.
Output: $SOSY$ is the sorted list of the elements in SY. Suppose $SOSY[j,k] = SY[l,q]$, then, $DATA[j,k]$ stores the ordered list $< SY[l,q], DAT[l,q] >$, where $DAT[l,q] =< q, l, SSF[l,q], SSD[l,q], SSP[l,q], SSE[l,q] >$. $DATA$ is IED stored in the processors $PR_i, PR_{i+1}, ..., PR_{i+(t-1)}$.

```
Begin /* W.l.g we assume $t$ is a power of two */
1. If ($t = 1$) then /* There is only one block */
   Processor $PR_i$ will do /* Solved sequentially */
   $(SOSY, SSP, SSD, SSE) = SEQGPC(SF, SY, Max_i)$.
   For $j = 1$ to $Max_i$ do
      If $SOSY[j,i] = SY[k,i]$ then
      $DATA[j,i] =< SY[k,i], < i, k, SSF[k,i], SSD[k,i],$
                  $SSP[k,i], SSE[k,i] >>$
   return($DATA$); exit.
2. Do Steps 2.1 and 2.2 in parallel /* $t > 1$ */
/* recursively solving left and right halves */
2.1 $(LDATA) = BlockGPC(SF, SY, i, \frac{t}{2})$
2.2 $(RDATA) = BlockGPC(SF, SY, i + \frac{t}{2}, \frac{t}{2})$
3. $(BPL, TF) = RANKP(LDATA, RDATA, i, t)$
/* From the definition of function RANKP we see that
$BPL[j,i]$ is a two tuple and the sequence consisting of
the first entries of all the tuples is the sorted list $SOSY$. */
4. If $SOSY[j,k] = SY[l,q]$, then $BPL[j,k] =< SY[l,q], dat >$
   From the function RANKP and definitions of $LDATA$
   and $RDATA$ we see that $TF[j,k]$ contains the necessary
   data as suggested in Lemma 1 for calculating $SSF[l,q]$,
   $SSD[l,q], SSP[l,q], SSE[l,q]$. Hence we compute
   $DATA[j,k] =< SY[l,q], < q, l, SSF[l,q], SSD[l,q],$
               $SSP[l,q], SSE[l,q] >$.
5. return($DATA$)
End.
```

Let $T_{comp}(t)$ be the computation time taken by $BlockGPC(SF, SY, i, t)$. From Theorem 3 $T_{comp}(1) = O(\frac{n}{p}\log(\frac{n}{p}))$ (Step 1 of the function). Step 4 takes $O(\frac{n}{p})$ computation time. From Theorem 2 we see that, Step 4 takes $O(\frac{n \log_2 t}{p} + \frac{\tau \log_2 t}{\sigma m \log_2(\frac{\tau}{\sigma m}+1)})$ time. Hence, $T_{comp}(t) = T_{comp}(\frac{t}{2}) + O(\frac{n \log_2 t}{p} + \frac{\tau \log_2 t}{\sigma m \log_2(\frac{\tau}{\sigma m}+1)}) = O(\frac{n}{p}(\log(\frac{n}{p}) + \log^2 t) + \frac{\tau \log_2^2 t}{\sigma m \log_2(\frac{\tau}{\sigma m+1})})$. Similarly, let $T_{comm}(t)$ be the communication time taken by *BlockGPCMerge*(SF, SY, i, t). $T_{comm}(1) = 0$. We can easily see that $T_{comm}(t) = T_{comm}(\frac{t}{2}) + O((\tau + \sigma m\lceil\frac{n}{pm}\rceil)\log t) = O((\tau + \sigma m\lceil\frac{n}{pm}\rceil)\log^2 t)$. Hence, the following theorem.

Theorem 4 *Function BlockGPC*(SF, SY, i, t) *takes* $O(\frac{n}{p}(\log(\frac{n}{p}) + \log^2 t) + \frac{\tau \log_2^2 t}{\sigma m \log_2(\frac{\tau}{\sigma m}+1)})$ *computation time and* $O((\tau+\sigma m\lceil\frac{n}{pm}\rceil)\log^2 t)$ *communication time.*

Function *BDSMGPC*$(F, Y) : (SD)$

Input: Two input sequences $f = (f(1), ..., f(n)$ and $y = (y(1), ..., y(n)$, each *IED* stored on the arrays F and Y respectively.

Output: The sequence $d = (d(1), d(2), ..., d(n))$ the general prefixes as defined in Definition 2, *IED* stored on the array D.

```
Begin
1. $DATA = BlockGPC(F, Y, 0, p)$
2. For every processor $PR_i$, $0 \leq i \leq p-1$ do in Parallel
     For $1 \leq j \leq n/p$ do
       Let $< a_1, < a_2, a_3, b_1, b_2, b_3, b_4 >> = DATA[j, i]$
/* $a_1 = SY[u, v]$, $a_2 = v$ and $a_3 = u$ and $b_2 = D[u, v]$ */
       Let $A[j, i] = < a_2, a_3, b_2 >$
3. $(A', c) = RANDOMROUTE(A)$.
/* In the above call $c = 1$ as exactly $\frac{n}{p}$ data are
destined for any processor (refer to Theorem A.1) */
4. For every processor $PR_i$, $0 \leq i \leq p-1$ do in Parallel
     For $1 \leq j \leq cMax_i$ do
       Let $< a_1, a_2, a_3 > = A'[j, i]$
       $D(a_2, a_1) = a_3$
End.
```

The above function and Theorems A.1 and 4 imply the following theorem.

Theorem 5 *Function BDSMGPC*(F, Y) *takes* $O(\frac{n}{p}(\log(\frac{n}{p}) + \log^2 p) + \frac{\tau \log_2^2 p}{\sigma m \log_2(\frac{\tau}{\sigma m}+1)})$ *expected computation time and* $O((\tau+\sigma m\lceil\frac{n}{pm}\rceil)\log^2 p)$ *expected communication time.*

5 Applications

5.1 Sorting

The following transformation is given in [6]. Let $x(1), x(2), ..., x(n)$ be the n numbers to be sorted. Set $y(i) = x(i)$, $f(i) = 1$, $\rho = \leq$ and $* = +$ (summation) in Definition 2 and we perform the *GPC* to compute D_i, $1 \leq i \leq n$. Clearly the prefix D_i gives the number of elements j such that $x(j) \leq x(i)$ and $j < i$. Again set $y(i) = x(n-i)$, $f(i) = 1$, $\rho = <$ and $* = +$ (summation) and we perform the *GPC* to compute D'_i, $1 \leq i \leq n$. Clearly the prefix D'_{n-i} gives the number of elements j such that $x(j) < x(i)$ and $j > i$. Now $D_i + D'_{n-i} + 1$ gives the position /of x_i in the sorted list. We now use the function Randomroute to route the x_i's such that the sorted list is *IED* stored on the array, say S, on a p-processor *BDM*. This and Theorem 5 imply part of Theorem 6.

Problem	$f(i)$	$y(i)$	ρ	$*$	Result
$2D$-maximal	$p_i[2]$	i	$<$	max	p_i is max. if $p_i[2] > D_i$
$3D$-maximal	$p_i[3]$	$p_i[2]$	$<$	max	p_i is max. if $p_i[3] > D_i$
$ECDF$-search	1	$a_i[2]$	$<$	$+$	D_i

Figure 2: Mapping of Dominance and Searching Problems onto *GPC*.

5.2 Computational Geometry

Given a set $S = \{p_1, p_2, ..., p_n\}$ of n points in D-dimensional space. A point p_i **D-dominates** a point p_j if and only if $p_i[k] > p_j[k]$, for all $1 \leq k \leq D$, where $p[k]$ denotes the k^{th} coordinate of a point p. The 2-dimensional **ECDF searching** problem consists of computing for each p in a 2-dimensional point set S, the number of points of S 2-dominated by p. The **D-dimensional maximal elements** problem is to determine the points in a given D-dimensional point set S, which are D-dominated by no other point. For each of the problems we sort the points in S by first coordinate and denote the lists sorted in ascending and descending order by $a_1, a_2, ..., a_n$ and $b_1, b_2, ..., b_n$ respectively. Figure 2 shows the transformation for the above mentioned problems [6].

Given two point sets A and B, the 2-set dominance problem in two dimensions is the problem of counting for each point p from B the number of points from A that p 2-dominates. Let $\mid A \mid= n$ and $\mid B \mid= m'$. Let $a_1, a_2, ..., a_{m'+n}$ be the list of the $m' + n$ points belonging to the set $A \cup B$, sorted in ascending order of their first coordinate. If $a_i \in A$, then, $f(i) = 1$ else $f(i) = 0$. $y(i) = a_i[2]$, $\rho = <$ and $* = +$. The result is D_i, i.e., point $a_i \in B$ 2-dominates D_i number of points in A [6]. Hence the following theorem.

Theorem 6 *Sorting problem, ECDF Searching problem, 2D and 3D maximal elements problem and 2-set Dominance Counting problem can be solved in* $O(\frac{n}{p}(\log(\frac{n}{p}) + \log^2 p))$ *expected computation time and* $O((\tau + \sigma m\lceil\frac{n}{pm}\rceil)\log^2 p)$ *expected communication time on a p-processor DSM system using the BDM model.*

5.3 Graph Theory

Permutation Graphs: Given a permutation π of the integer set V = $\{1, 2, 3, ..., n\}$, π defines a graph G_p = (V,E), where $(i, j) \in$ E if and only if $(i-j)*(\pi^{-1}(i) - \pi^{-1}(j)) < 0$, where $\pi^{-1}(k)$ denotes the position of k in π. A graph G is a permutation graph if and only if it is isomorphic to some G_π . We assume that the permutation π is *IED* stored in the array P and the inverse permutation π^{-1} is *IED* stored in the array *PI*. Given P it is straightforward to compute *PI* on a *DSM* system using the function *RandomRoute*. The arrays *LSV, MSV, CNUM, W* and V defined in [1] can be computed using the function *BDPRECOMP*. This, Theorem A.2 and the algorithms CONCOMP, Cutvertices, Bridges presented in [1] imply a part of Theorem 7.

Binary Tree Reconstruction: We assume that the preorder traversal *pre* and inorder traversal *in* are

IED stored in the array *PRE* and *IN* respectively. It is straightforward to implement the algorithm presented in [5] using the functions *BDPRECOMP* and *RandomRoute* to reconstruct the binary tree represented by these traversals. Hence the following Theorem.

Theorem 7 *The problems of finding Connected Components, Cut Vertices and Bridges of a given Permutation graph of n vertices and the problem of reconstructing a binary tree from its Inorder and Preorder traversals can be solved in* $4\tau\lceil\frac{\log_2 p}{\log_2(\frac{\tau}{\sigma m}+1)}\rceil + \tau + \sigma m + O(\frac{n}{p})$ *expected communication time and* $O(\frac{n}{p} + \frac{\tau \log_2 p}{\sigma m \log_2(\frac{\tau}{\sigma m}+1)})$ *expected computation time on a p-processor DSM system using the BDM model.*

6 Conclusion

In this paper we have presented an efficient algorithm for *GPC* on *DSM* systems using the *BDM* model with applications. Since sorting is transformed into an instance of *GPC*, *GPC* has a lower bound of $\Omega(n \log n)$ at least for some specific choice of operators $*$. As seen before, if $n > p \log p$, our algorithm for *GPC* takes $O(\frac{n \log n}{p})$ computation time. Hence, we have achieved optimal speed up in computation for the sorting problem for such realistically larger values of n. Further, our method is conceptually simpler than the algorithms presented in [3] for sorting. From the analytical expressions for the time taken by *BDPRECOMP* and *BlockMerge*, we see that the time taken will increase with the number of processors (p) for smaller size of the inputs. From Corollary 1 we see that the computation time of *BlockMerge* is $O(n \log_2 p/p)$ and hence we cannot expect any scalability from $p = 2$ to $p = 4$. It is interesting to note that the timing measurements agree with the theoretical predictions and the graphical plots illustrates the scalability (refer Figure 3). It is worth noting that the merge function is recursive and it executes on a DMS which might serve as a pointer for incorporating parallel recursion in automatic parallelising compilers.

Appendix A

Function *RANDOMROUTE*$(A) : (A', c)$

Input: $A[1 : \lceil\frac{n}{p}\rceil, 0 : p-1]$ is the input array *IED* stored on a p-processor *BDM* machine, such that each element of A consists of a packet $(i, data_i)$ of constant size, where i is the index of the processor to which $data_i$ has to be routed. α is a constant such that no processor is the destination of more than $\alpha\lceil\frac{n}{p}\rceil$ elements on the whole.

Output: $A'[1 : c\lceil\frac{n}{p}\rceil, 0 : p-1]$ is the output array holding the routed data, *IED* stored on a p-processor *BDM* machine, such that all the data with the processor PR_i, $0 \leq i \leq p-1$, as the destination will be available in one of the locations $A'[j, i]$, $1 \leq j \leq c\lceil\frac{n}{p}\rceil$ in the processor PR_i, where c is larger than $\max\{1+\frac{1}{\sqrt{2}}, \alpha+\frac{\sqrt{\alpha}}{2}\}$. The function stores a copy of c in every processor PR_i, $0 \leq i \leq p-1$.

The function is implemented using the **Randomized_Routing** algorithm suggested in [3].

Theorem A.1 *Function* **RANDOMROUTE**(A) *completes within* $2(\tau + c\lceil\frac{n}{p}\rceil)$ *communication time and* $O(c\lceil\frac{n}{p}\rceil)$ *computation time with high probability, where c is larger than max*$\{ 1 + \frac{1}{\sqrt{2}}, \alpha + \frac{\sqrt{\alpha}}{2} \}$, $p^2 < \frac{n}{6 \ln n}$ □

Function *BDPRECOMP*$(A, r, \bigtriangledown) : (A')$.

Input: Given a sequence of ordered pairs $< a_1, data_1 >, < a_2, data_2 >, \cdots, < a_{t(n/p)}, data_{t(n/p)} >$, *IED* stored on the array A, on t processors of a *BDM* machine, $data_i$ is the data (if any) associated with a_i, $1 \leq i \leq t(n/p)$. $\bigtriangledown$ is a binary associative operator $\in \{+, Min, Max, ...\}$.

Output: A sequence of ordered pairs $< a'_1, data'_1 >, < a'_2, data'_2 >, \cdots, < a'_{t(n/p)}, data'_{t(n/p)} >$, *IED* stored on the array A', on a p-processor *BDM* machine, where, $a'_i = \bigtriangledown_{k=1}^{i} a_k$ and $data'_i$ is the data associated with a'_i (if any).

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

Theorem A.2 ([4]) *Given a sequence* $(a_1, a_2, ..., a_{t(n/p)})$ *of numbers IED stored on t processors of a BDM machine, we can compute the prefix sums* $ps_i = \sum_{j=1}^{i} a_j$, $1 \leq i \leq r$, *in* $4\tau\lceil\frac{\log_2 t}{\log_2(\frac{\tau}{\sigma m}+1)}\rceil + \tau + \sigma m$ *communication time and* $O(\frac{n}{p} + \frac{\tau \log_2 t}{\sigma m \log_2(\frac{\tau}{\sigma m}+1)})$ *computation time. This complexity holds for prefix maxima, prefix minima and similar associative operators.*

References

[1] K. Arvind, V. Kamakoti, and C. Pandurangan. Efficient parallel algorithms for permutation graphs. *Journal of Parallel and Distributed Computing*, 26:116–124, 1995.

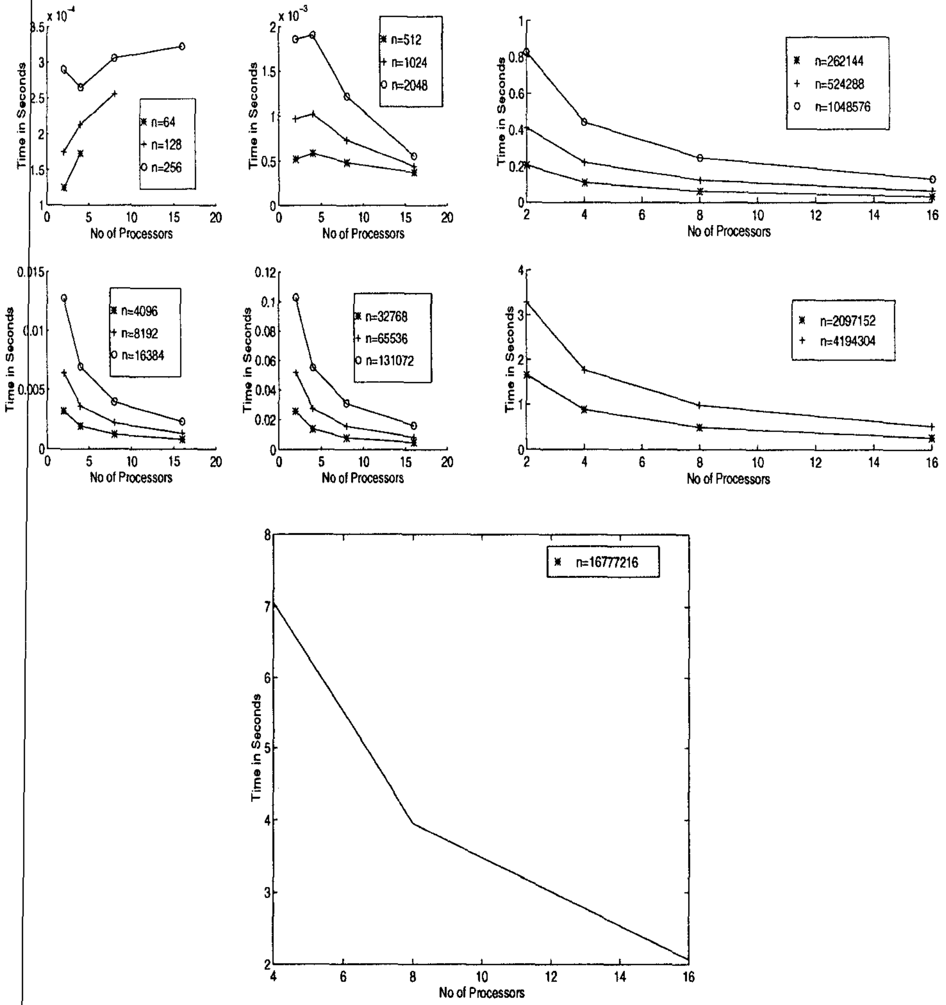
[2] K. E. Batcher. Sorting networks and their applications. In *Proceedings AFIPS 32nd Spring Joint Computer Conference*, pages 307–314, 1968.

[3] F. J. Jaja and K. W. Ryu. The block distributed memory model. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):830–840, 1996.

[4] V. Kamakoti and N. Balakrishnan. Efficient randomized algorithm for the closest pair problem on distributed shared memory systems. Report, SERC, IISc, Bangalore - 560 012, India, 1996.

[5] V. Kamakoti and C. Pandurangan. An optimal algorithm for reconstructing a binary tree. *Information Processing Letters*, 42(2):113–115, 1992.

[6] F. Springsteel and I. Stojmenovic. Parallel general prefix computations with geometric, algebraic and other applications. Report, University of Ottawa, Computer Science, Ottawa, Ontario K1N 9B4, Canada.



(a). Prefix Sum on IBM SP2

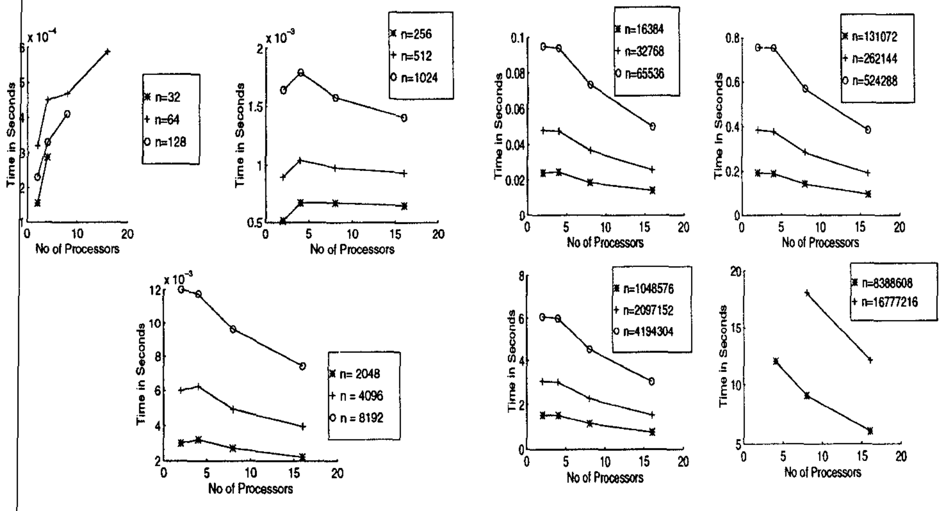


Figure 3: (b). Block Merging on IBM SP2