

# PACK/UNPACK on Coarse-Grained Distributed Memory Parallel Machines

Seungjo Bae\* and Sanjay Ranka†

*\*Corresponding author*

2-120 Center for Science and Technology  
School of Computer and Information Science  
Syracuse University  
Syracuse, NY 13244-4100  
Phone: (315) 469-9211  
Email: [sbae@top.cis.syr.edu](mailto:sbae@top.cis.syr.edu)

†CSE Building, #301  
Department of Computer Science  
University of Florida  
Gainesville, FL 32611  
Phone: (352) 392-1526  
Email: [ranka@cis.ufl.edu](mailto:ranka@cis.ufl.edu)

Will appear in *Special Issue of Journal of Parallel and Distributed Computing on  
“Compilation Techniques for Distributed Memory Systems”*

### **Abstract**

PACK/UNPACK are Fortran 90/HPF array construction functions that derive new arrays from existing arrays. We present algorithms for performing these operations on coarse-grained parallel machines. Our algorithms are relatively architecture independent and can be applied to arrays of arbitrary dimensions with arbitrary distribution along every dimension. Experimental results are presented on the CM-5.

# List of symbols

---

Symbol	Name
$\Delta$	Delta
$\Pi$	Pi
$\Sigma$	Sigma
$\Theta$	Theta
$\alpha$	alpha
$\beta$	beta
$\wedge$	logical and
$\epsilon$	epsilon
$\eta$	eta
$\lceil \rceil$	ceiling
$\delta$	delta
$\forall$	forall
$\gamma$	gamma
$\leftarrow$	leftarrow
$\mu$	mu
$\ni$	such that
$\tau$	tau
$\zeta$	zeta

---

## **Biography**

### **Seungjo Bae**

I am currently a Ph.D. candidate in computer science at Syracuse University, New York. My research interests are runtime support for data-parallel languages, parallel compilers, the design and analysis of parallel algorithms, and high-performance computing. I received an M.S. degree in computer science from Syracuse University in 1992 and a B.S. degree in computer science from Yonsei University, Seoul, Korea, in 1987.

### **Sanjay Ranka**

# 1 Introduction

PACK/UNPACK are the array construction functions that derive new arrays from existing arrays [2, 3]. These functions are part of the transformational intrinsic functions in FORTRAN 90, CM FORTRAN that were also incorporated into HPF [2, 3, 5]. PACK gathers selected elements from an input array of any rank under the control of a logical mask array and constructs the new vector (rank-one array). UNPACK scatters all elements of an input vector to the array of any rank under the control of a mask array.

The parallel PACK/UNPACK algorithm consists of two stages:

1. **Ranking**: Rank all *true* values of mask array  $M$ .
2. **Redistribution**: Redistribute all corresponding elements of input array  $A$  (or input vector  $V$  in UNPACK) to result vector  $V$  (or result array  $A$ ).

In this paper we present several algorithms for performing the ranking in parallel. Our algorithms are relatively architecture independent and can be applied to arrays of arbitrary dimensions with arbitrary distribution along every dimension. In the second stage we redistribute selected elements of an array among all processors. This stage requires many-to-many personalized communication.

The rest of the paper is organized as follows. The coarse-grained parallel machine model used in our paper is presented in Section 2 and definitions and notations used in this paper are explained in Section 3. In Section 4 we describe the parallel PACK/UNPACK algorithm using the parallel ranking algorithm, and we present the parallel ranking algorithm in Section 5. We explain the optimized schemes for PACK/UNPACK in Section 6. Section 7 presents the experimental results, and we state our conclusions in Section 8.

## 2 Coarse-Grained Parallel Machine Model

A coarse-grained parallel machine consists of several processors that have a private local memory system and are connected by an interconnection network. We assume a two-level model of computation rather than any specific underlying network. In the two-level model we assume the cost for a remote-memory access is fixed and that it is independent of the distance between a sender and a receiver. We also assume that the cost for unit local computation is  $\delta$ . The cost for communication between two processors consists of the start-up cost,  $\tau$ , and the data-transfer rate,  $1/\mu$ . We assume that  $\tau$  and  $\mu$  are constant and independent of the link congestion and distance between two processors.

It follows that the underlying interconnection network can be regarded as a virtual crossbar network in the two-level model. These assumptions closely model the behavior of the CM-5 on which our experimental results are presented. Although our algorithms are analyzed under these assumptions, most of them are architecture-independent and can be efficiently implemented on meshes and hypercubes with

wormhole routing. Under the assumption that there is no node contention, the time taken to send a message from one processor to another is modeled as  $\tau + \mu m$ , where  $m$  is the size of the messages [8].

### 3 Definitions and Notations

This section defines the notations and definitions that will be used throughout the paper. PACK gathers elements of an input array,  $A$ , to a result vector,  $V$ , under the control of a mask array,  $M$  where  $A$  has a rank of  $d$  and a shape of  $(N_{d-1}, N_{d-2}, \dots, N_1, N_0)$ . We assume that  $A$  is distributed among the logical processors  $Pn(P_{d-1}, P_{d-2}, \dots, P_1, P_0)$  using  $\text{block-cyclic}(W_{d-1}, W_{d-2}, \dots, W_1, W_0)$  partitioning where  $P_i$  and  $W_i$  ( $1 \leq W_i \leq N_i/P_i$ ) are the number of processors and the block size on dimension  $i$ , respectively. We also assume that  $M$  is conformable with and aligned to  $A$ , and the size of  $V$  is the same as the number of *true* values in  $M$  (say *Size*).

For the sake of simplicity, we assume that for all  $i$  ( $0 \leq i < d$ ),  $P_i \setminus N_i, W_i \setminus N_i$  and  $P_i W_i \setminus N_i$ . Also, we assume that the row-major indexing scheme is used and all indices start from zero. Hence, if all elements of mask array  $M$  have the value *true*, an array element  $A(i_{d-1}, \dots, i_0)$  has the rank  $\sum_{l=0}^{d-1} \left( i_l \prod_{k=0}^{l-1} N_k \right)$ . The other notations used in this paper are:

- $P$ : Total number of processors where  $P = \prod_{i=0}^{d-1} P_i$ .
- $N$ : Global array size where  $N = \prod_{i=0}^{d-1} N_i$ .
- $L_i$ : Local array size on dimension  $i$  where  $L_i = N_i/P_i = T_i W_i$  and  $0 \leq i < d$ .
- $L$ : Local array size where  $L = N/P = \prod_{i=0}^{d-1} L_i$ .
- $S_i$ : Tile<sup>1</sup> size where  $S_i = P_i W_i$ .
- $T_i$ : Total number of tiles. This is the same as the total number of blocks assigned to a processor where  $T_i = N_i/(P_i W_i) = N_i/S_i = L_i/W_i$ .

The above definitions and notations presented for PACK are also valid for UNPACK, except for the following. UNPACK scatters elements of an input vector,  $V$ , to an array,  $A$ , under the control of a mask array,  $M$ . We assume that  $V$  has a shape of  $(N')$  and is distributed, using  $\text{block-cyclic}(W')$ , among the logical processors  $Pn(P')$ . Note that  $N'$  should not be less than *Size*. We also assume that a field array,  $F$ , and a result array,  $A$ , are conformable with and aligned to  $M$ .

### 4 Parallel PACK/UNPACK Algorithm

The parallel PACK/UNPACK algorithm consists of two stages, ranking and redistribution. In the first stage we rank all selected elements under the control of mask array  $M$  by applying the parallel ranking

---

<sup>1</sup>On dimension  $i$ , a tile consists of  $P_i$  continuous blocks of size  $W_i$ . Hence one tile is mapped to  $P_i$  processors.

algorithm, which will be presented in Section 5. In the following subsections we present an algorithm for PACK/UNPACK. Also, several optimization schemes will be presented in Section 6.

#### 4.1 PACK

After the ranking stage, we know the rank of each element to be gathered in result vector  $V$  and  $Size$ , which is also the size of  $V$ . For the sake of simplicity we assume that  $V$  is distributed in block partitioning among  $P$  processors. Then the rank (say  $r$ ) of each selected element of  $A$  is the same as the global index of the element of  $V$  such that the element of  $A$  is transferred to  $V[r]$ .

For the redistribution stage, many-to-many personalized communication is required. Each processor knows where the local selected elements of  $A_l$  have to be sent, that is, which is the receiving processor. Also, all messages with the same destinations may be coalesced to decrease the communication overhead.

#### 4.2 UNPACK

For all  $r$  ( $0 \leq r < Size$ ),  $V[r]$  should be transferred to the element of  $A$  whose corresponding element of  $M$  has the rank  $r$  after the ranking stage. Note that the transfer from field array  $F$  to  $A$  is local computation when the corresponding mask element is *false*.

In the redistribution stage, no processor knows who needs the data, that is, which is the receiving processor. Therefore, two-stage communication is required. Each processor first sends a request to each sender, and then each processor receiving any request sends data back to the receiver. It follows that the communication time for the second stage of UNPACK may be two times as large as that for PACK.

### 5 Parallel Ranking Algorithm

In this section we present a parallel ranking algorithm. We apply this algorithm to mask array  $M$  in PACK/UNPACK. The rest of the presentation is limited to PACK for obvious reasons. First of all, we define a local input array and a local mask array,  $A_l$  and  $M_l$ , of shape  $(L_{d-1}, \dots, L_0)$ .

The ranking algorithm computes the rank of each element to be packed using vector prefix-sum and reduction-sum operations, which will be explained in Section 5.1, without actually moving any elements among processors. The ranking algorithm consists of the following three steps:

1. *Initial step (local scan)*: find local elements to be packed.
2. *Intermediate step*: compute the intermediate rank of the selected elements dimension-by-dimension by using the *prefix-reduction-sum*.
3. *Final step (ranking)*: rank all selected elements.

We use two local arrays,  $PS_i$  and  $RS_i$ , of shape  $(L_{d-1}, L_{d-2}, \dots, L_{i+1}, T_i)$  to compute the intermediate rank for each dimension  $i$  ( $0 \leq i < d$ ). We thus need a total of  $2d$  local arrays, each of which can be regarded as a base-rank array on each dimension. Figure 1 shows an example of a parallel ranking algorithm on a one-dimensional array in which the input array,  $A(16)$ , and the mask array,  $M(16)$ , are distributed in block-cyclic(2) on four processors.

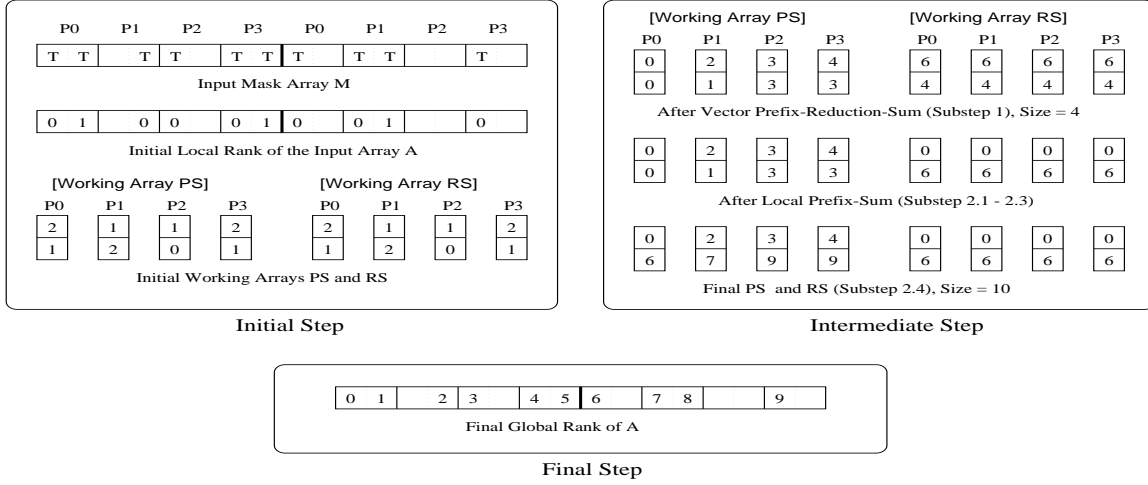


Figure 1: Example of parallel ranking algorithm on a one-dimensional array

## 5.1 Vector Prefix-Reduction-Sum Operation

Assume that each processor,  $P_i$ , has a local vector,  $V_i[0 : M - 1]$ . Let  $P$  be the total number of processors. Then the *vector reduction-sum* computes an element-wise sum of the local vector located in each processor. The result vector,  $R[0 : M - 1]$ , is stored in all the processors. Hence

$$R[j] = \sum_{i=0}^{P-1} V_i[j] \text{ where } 0 \leq j < M.$$

The *vector prefix-sum* computes an element-wise exclusive prefix sum of the local vector located in each processor. Each processor,  $P_i$ , has the result vector  $F_i[0 : M - 1]$ :

$$F_i[j] = \sum_{k=0}^{i-1} V_k[j] \text{ where } 0 \leq j < M.$$

Note that processor  $P_0$  has the result vector  $F_0[j] = 0$  for all  $j$  ( $0 \leq j < M$ ).

The basic structure of the algorithm for the prefix-sum is the same as that for the reduction-sum. Therefore, if the same input vector is provided for both communication primitives, we can combine two primitives in order to reduce the communication overhead, especially the total amount of the start-up cost. Now we define the communication primitive, *prefix-reduction-sum*, which performs vector

prefix-sum and reduction-sum simultaneously where the prefix operation is exclusive.<sup>2</sup>

For the vector prefix-reduction-sum, two algorithms may be used: *direct algorithm* and *split algorithm* [1, 6]. We first define the cost function for prefix-reduction-sum as  $\text{PRS}(P, M)$ . Then the cost function  $\text{PRS}(P, M)$  is defined as  $\mathcal{O}(\tau \lg P + \mu M)$  in the split algorithm and  $\mathcal{O}((\tau + \mu M) \lg P)$  in the direct algorithm, where  $\tau$  is the startup time and  $\mu$  is the per-element transfer time [1, 6]. Thus, as the number of processors and the vector size increase, the split algorithm will give much better performances than the direct algorithm does. Readers are referred to [1, 6] for detailed algorithms.

## 5.2 Initial Step (Local Scan)

We first define *slice*, which is a portion of a local array,  $M_l$  (or  $A_l$ ), such as  $M_l(:, \dots, :, iW_0 : (i+1)W_0 - 1)$  where  $0 \leq i < T_0$ . Then we may regard  $M_l$  (or  $A_l$ ) as a sequence of  $\left(\prod_{i=1}^{d-1} L_i\right) T_0$  slices where the size of a slice is equal to the block size on dimension 0,  $W_0$ . For example, the first and the last slices are  $M_l(0, \dots, 0, 0 : W_0 - 1)$  and  $M_l(N_{d-1} - 1, \dots, N_1 - 1, (T_0 - 1)W_0 : T_0W_0 - 1)$ , respectively.

Now we scan all element of  $M_l$  slice by slice. At the same time, for each element to be packed we set the initial rank that is valid in a slice and initialize  $PS_0$  and  $RS_0$ . At the end of this step,  $PS_0(:, \dots, :, i)$  and  $RS_0(:, \dots, :, i)$  are the total number of local packed elements in slice  $A_l(:, \dots, :, iW_0 : (i+1)W_0 - 1)$  where  $0 \leq i < T_0$ . This initial step takes time  $\Theta\left(\prod_{i=0}^{d-1} L_i\right)$ .

## 5.3 Intermediate Step

The intermediate step consists of  $d$  (rank of the input array) steps. At each intermediate step  $i$  ( $0 \leq i < d$ ), initialized  $PS_i$  and  $RS_i$  are provided from the previous intermediate step  $i - 1$ . Each intermediate step  $i$  consists of three sub-steps:

- Substep 1: Vector prefix-reduction-sum on  $PS_i$  and  $RS_i$  along dimension  $i$ .
- Substep 2: Local prefix-sum on  $RS_i$ . This is followed by using modified  $RS_i$  to modify  $PS_i$ .
- Substep 3: Initialization of  $PS_{i+1}$  and  $RS_{i+1}$  for the next intermediate step  $i + 1$ .

Each substep is explained in detail in Figure 2.

If input array  $A$  of rank  $d$  is regarded as a one-dimensional array, then the input array can be partitioned into contiguous sub-arrays such that each sub-array has a uniform shape and is non-overlapping. The main idea of the ranking algorithm is that we compute the rank valid in each sub-array while we enlarge the sub-array at each step. Each enlargement can be done by vector prefix-reduction-sum or local prefix-sum. At the end of the algorithm there will be only one sub-array identical to input array  $A$ .

---

<sup>2</sup>If a control network is available, as on the CM-5, we do not need to combine reduction-sum with prefix-sum. Each primitive can be performed in  $\mathcal{O}(M)$  time.

As the algorithm proceeds in intermediate step  $i$  ( $0 \leq i < d$ ), we can compute the ranks valid in each sub-array of size  $\Delta$  such as  $\Delta = [1 \times \cdots \times 1 \times \delta_{i+1} \times \delta_i \times N_{i-1} \times \cdots \times N_0]$  where  $\delta_i = W_i, S_i$  or  $N_i$  and  $\delta_{i+1} = 1$  or  $W_{i+1}$ . Figure 2 shows how the sub-array is enlarged after each sub-step in intermediate step  $i$ . For each element to be packed we can figure out how many *true* elements are ahead of it in each sub-array by summing all  $PS_j$  ( $0 \leq j \leq i$ ).<sup>3</sup> But, note that such summation is not actually performed in the intermediate step. It will be explained later how to combine arrays of different shapes in the final step. Note that the initial  $\Delta$  is given by  $[1 \times \cdots \times 1 \times W_0]$  after the initial (local scan) step.

## 5.4 Final Step

In the final step we sum all base-rank arrays  $PS_i$  ( $0 \leq i < d$ ) through  $d - 1$  summations such that  $PS_i \leftarrow PS_i + PS_{i+1}$  ( $0 \leq i < d - 1$ ). Two arrays of different shapes,  $PS_i(L_{d-1}, L_{d-2}, \dots, L_{i+2}, L_{i+1}, T_i)$  and  $PS_{i+1}(L_{d-1}, L_{d-2}, \dots, L_{i+2}, T_{i+1})$ , can be added as follows:

$$\begin{aligned} \forall j, k \ni kW_{i+1} \leq j < (k+1)W_{i+1} \wedge 0 \leq k < T_{i+1}, \\ PS_i(:, \dots, :, j, :) \leftarrow PS_i(:, \dots, :, j, :) + PS_{i+1}(:, \dots, :, k). \end{aligned}$$

The final resulting array after  $d - 1$  summation is regarded as a final base-rank array,  $PS_f$ .

Let  $x$  be  $A_l(i_{d-1}, \dots, i_0)$  with *true* mask value where  $0 \leq i_k < L_k$  and  $0 \leq k < d$ . Then the final rank of  $x$  can be computed:  $rank(x) = initial\text{-}rank \text{ of } x + PS_f(i_{d-1}, \dots, i_1, i_0 \text{ div } W_0)$ . Note that  $i_0 \text{ div } W_0$  indicates the tile number on dimension 0 to which  $x$  belongs. Now  $\Delta$  is the same as the size of the whole array. This final step takes time  $\mathcal{O}\left(\left(\prod_{i=1}^{d-1} L_i\right) T_0 + \alpha\right)$  where  $\alpha$  is the maximum number of local packed elements among all processors ( $0 \leq \alpha \leq \prod_{i=0}^{d-1} L_i$ ).

## 6 Optimization Schemes for PACK/UNPACK

In this section we present two optimized schemes: *compact storage scheme* and *compact message scheme*. We also present an optimized method using redistribution for the input array that is distributed cyclically in order to reduce the ranking overhead.

### 6.1 Compact Storage Scheme

We first present *simple storage scheme* and then explain the optimized scheme, *compact storage scheme*, for reducing local computation time. In the simple storage scheme we save information regarding local packed elements during the initial step of the ranking stage. This information includes a local index on each dimension, a tile number,<sup>4</sup> and an initial local rank valid in a slice for each local packed element. This information saved during the initial step can be used in the final step of the ranking stage. During

<sup>3</sup>For this reason we may call the final  $PS_i$  at the end of intermediate step  $i$  a base-rank array on dimension  $i$  ( $0 \leq i < d$ ).

<sup>4</sup>A tile number will be used to access the final base-rank array when the final rank is computed in the final step.

---

**Input from the Intermediate Step  $i - 1$** 

1. Initialized  $PS_i(L_{d-1}, \dots, L_{i+1}, T_i)$  and  $RS_i(L_{d-1}, \dots, L_{i+1}, T_i)$ .
2.  $\Delta = [1 \times \dots \times 1 \times W_i \times N_{i-1} \times \dots \times N_0]$ .

**Substep 1: Vector prefix-reduction-sum**

1. Vector prefix-reduction-sum on  $PS_i$  (exclusive prefix sum) and  $RS_i$  (reduction sum) along the dimension  $i$ .
2.  $\Delta = [1 \times \dots \times 1 \times S_i \times N_{i-1} \times \dots \times N_0]$ .

**Complexity:**  $\Theta \left( \text{PRS} \left( P_i, \left( \prod_{k=i+1}^{d-1} L_k \right) T_i \right) \right)$ .

**Substep 2: Local Prefix Sum**

1. If  $0 \leq i < d - 1$ ,  $RS_{i+1}(:, \dots, :, k) \leftarrow RS_i(:, \dots, :, l, T_i - 1)$  where  $0 \leq k < T_{i+1}$  and  $l = (k + 1)W_{i+1} - 1$ . In step  $d - 1$ ,  $Size \leftarrow RS_{d-1}(T_{d-1} - 1)$  where  $Size$  is the total number of packed elements.
2. If  $0 \leq i < d - 1$ , set the starting point of each segment (of total  $T_{i+1}$  segments) at  $RS_i(:, \dots, :, m, 0)$  for all  $m$  such that  $m \bmod W_{i+1} = 0$ . Note that in step  $d - 1$ , there is only one segment.
3. Segmented exclusive prefix-sum on  $RS_i$  using the same indexing scheme.
4.  $PS_i \leftarrow PS_i + RS_i$ .
5.  $\Delta = [1 \times \dots \times 1 \times W_{i+1} \times N_i \times N_{i-1} \times \dots \times N_0]$ .

**Complexity:**  $\Theta \left( \left( \prod_{k=i+1}^{d-1} L_k \right) T_i \right)$

**Substep 3: Modify  $PS_{i+1}$  and  $RS_{i+1}$** 

1. If  $0 \leq i < d - 1$ ,  $PS_{i+1}(:, \dots, :, k)$  and  $RS_{i+1}(:, \dots, :, k) \leftarrow RS_{i+1}(:, \dots, :, k) + PS_i(:, \dots, :, l, T_i - 1)$  where  $0 \leq k < T_{i+1}$  and  $l = (k + 1)W_{i+1} - 1$ . In step  $d - 1$ ,  $Size \leftarrow Size + RS_{d-1}(T_{d-1} - 1)$

**Complexity:**  $\Theta \left( \left( \prod_{k=i+2}^{d-1} L_k \right) T_{i+1} \right)$

Figure 2: Intermediate step  $i$  ( $0 \leq i < d$ )

---

the final step we first compute the final base-rank array  $PS_f$ . We then calculate the final rank for each local packed element by using  $PS_f$  while we scan the information saved in the initial step. In addition, we generate the communication vector (say  $sendl$ ) on each sending processor.<sup>5</sup> This  $sendl$  vector will be used in the redistribution stage. The advantage of this simple storage scheme is that programming is simple, but this scheme requires at least four memory-write and memory-read operations for each local packed element. As the rank of the input array increases, the amount of memory access will increase.

Alternatively, to reduce local memory access time we do not save any information at all for local packed elements. First of all, we need to copy  $PS_0$  to a counter array,  $PS_c$ , at the end of the initial step. After computing  $PS_f$  we can make the  $sendl$  vector by comparing each element of  $PS_c$  with that of  $PS_f$  during the final step. Suppose the  $k^{th}$  element of  $PS_c$  is  $n$  ( $0 \leq n < W_0$ ). Then there are  $n$  local packed elements,  $e_0, e_1, \dots, e_{n-1}$ , in a slice,  $s_k$  ( $0 \leq k < \left(\prod_{i=1}^{d-1} L_i\right) T_0$ ). Note that the  $k^{th}$  element of  $PS_f$  gives the final global rank of  $e_0$ . Hence the global ranks of  $n$  packed elements in slice  $s_k$  are  $r_0, r_1, \dots, r_{n-1} \equiv r_0, r_0 + 1, \dots, r_0 + n - 2, r_0 + n - 1$ . Since we know the block size of the result vector, we can easily figure out the processor to which each packed element should be sent, comparing the  $k^{th}$  element of  $PS_c$  with that of  $PS_f$  for each slice  $s_k$ . Note that the size of two arrays is  $\left(\prod_{i=1}^{d-1} L_i\right) T_0$ , which depends on the distribution on dimension 0. Hence time for generating the  $sendl$  vector depends mainly on the number of tiles on dimension 0,  $T_0$ .

In the redistribution stage, message-coalescing requires another local scan. However, this local scan can be done more efficiently than that in the initial step. That is, we scan a slice only if there is at least one packed element in the slice. This can be done by checking the counter array,  $PS_c$ . There are two scanning methods for the scan of such a slice:

1. Scan a slice until we collect all packed elements (optimized method).
2. Scan the whole slice without checking whether we collect all packed elements or not, which is the same as in the initial step.

In our comparison of two scanning methods our experimental results showed the first scanning method was better, although the difference was not significantly large. Thus our implementation was based on the first method. But the second method will be used in Section 6.4.1 only for the purpose of analysis.

## 6.2 Compact Message Scheme

The redistribution stage of PACK is a *WRITE* operation. Therefore, we need to send not only the selected datum value but also its global rank, which is the same as the global address of the selected datum in the result vector. In the simple storage scheme the communication message consists of the

---

<sup>5</sup>The recognized destination processor may be added to the information for each local packed element.

pair (*datum value, global rank*). This scheme is simple but always requires a message twice as long as the number of packed elements.

To reduce communication overhead (message size), we alternatively use the *compact message scheme*. As we observed in the previous subsection, the global ranks of  $n$  packed elements,  $r_0, r_1, \dots, r_{n-1}$ , in slice  $s_k$ , are  $r_0, r_0 + 1, \dots, r_0 + n - 2, r_0 + n - 1$ . These  $n$  packed elements will be sent either to one processor or to several processors, according to the block size of the result vector. Therefore, we can partition these  $n$  values into  $i$  segments ( $1 \leq i < P$ ) according to their destination processor. Each segment will be sent to a different destination processor if the result vector is distributed in block. Since the global ranks for the packed elements in each segment are consecutively numbered, we do not need to send all global ranks together with the packed elements, but just the rank of the first element in a segment, the number of the element in a segment, and the packed elements. Therefore, in this optimized scheme the communication message consists of segments, with each segment having the form (*base-rank, number of data in the segment, datum, ..., datum*). As in the previous compact storage scheme, we can figure out the total number of segments for each slice,  $s_k$ , by comparing the  $k^{th}$  element of  $PS_c$  with that of  $PS_f$ .

Since the size of each segment is at least 3, this optimized scheme may not be useful if the slice size is 1 (i.e., the input array is distributed cyclicly on dimension 0) or if there is only one packed element in a slice. Note that we already assume that the result vector is distributed in block. If the result vector is not distributed in block, the number of segments will increase as the block size of the result vector decreases. Hence this optimized scheme would give the best results for a block-distributed result vector.

### 6.3 Cyclic to Block Distribution

The complexity of the ranking algorithm mainly depends on the value of  $T_i$  (number of tiles on dimension  $i$ ), especially the number of tiles on the lower dimension. Therefore, we have the lowest ranking overhead in block distribution and the highest ranking overhead in cyclic distribution. Also, as seen in the previous subsections, two optimization schemes are not useful when the input array is distributed cyclicly on dimension 0.

Consider an input array of rank  $d$  such that each dimension is distributed in either cyclic or block distribution. Suppose dimension  $i$  of the input array is distributed cyclicly. The ranking overhead can be minimized by redistributing the array along dimension  $i$  to the block distribution. This can be done for one or all dimensions. Once the redistribution is done as the preliminary step we can apply the ranking algorithm with the lowest ranking overhead, since all dimensions are distributed in block.

Note that this redistribution scheme will not be a feasible option for UNPACK. Since UNPACK is a *READ* operation, we should return result array  $A$  with the original distribution at the end of UNPACK. This may result in two steps of redistributions: one for  $M$  and  $F$  before performing UNPACK, and the

other for  $A$  before returning  $A$  at the end of UNPACK. The details of the communication detection algorithms for array redistribution from cyclic to block distribution can be found in [7]. In the following we describe two redistribution schemes as the preliminary steps in PACK.

**Redistribution of Selected Data** We redistribute selected input data to be packed, but no data from the mask array. When we perform message coalescing after communication detection, we decide which elements of the input array should be redistributed, that is, only an element whose mask value is true is sent to the target processor. On the receiver, first of all, we initialize all values of the temporary mask array to *false*. Then we store each received element in a temporary input array and set the corresponding element of the temporary mask array to *true*. Hence this scheme would be useful when the total number of packed elements is relatively small. Since we redistribute only selected elements, we need to send the global index together with the selected element. If the rank of the input array is larger than 1 (say  $r$ ), we may need to combine  $r$  indices into one global index to minimize the message size on the sender, and we may need to decompose the global index into  $r$  indices on the receiver.

**Redistribution of Whole Arrays** When the total number of packed elements is relatively large, it is more useful to redistribute the input array and the mask array in order to reduce the local computation overhead entailed in the previous redistribution scheme. But it should be noted that we need two phases of communication detection in this scheme [7]: one for elements to be sent and the other for those to be received on each processor. Therefore, if the total costs for redistribution are dominated by the cost of communication detection, the performance of this scheme would not be better than that for the previous scheme. As in the previous scheme, it would be better to use two temporary arrays for  $A_l$  and  $M_l$ . If not, we need to redistribute  $A$  and  $M$  again before returning result vector  $V$  at the end of PACK.

## 6.4 Modeling for Local Computation

In Section 5 we presented the analysis for the ranking algorithm, which excluded the analysis for the storage scheme in the ranking stage. To compare the performances of the three schemes, *simple storage scheme* (SSS), *compact storage scheme* (CSS), and *compact message scheme* (CMS), we present the analysis for local computation, focusing on the storage scheme and the message composition scheme. First, we define notations:

- $E_i$ : Total number of local elements to be packed on processor  $i$  where  $0 \leq E_i \leq N/P$  and  $0 \leq i < P$ . Hence,  $Size = \sum_{i=0}^{P-1} E_i$ .
- $E_a$ : Total number of local packed elements assigned to each processor after the PACK operation is finished, that is,  $E_a = \lceil Size/P \rceil$ .

- $C$ : Total number of slices assigned to each processor  $i$ , where  $C = \left(\prod_{j=1}^{d-1} L_j\right) T_0$  and  $0 \leq i < P$ .
- $Gs_i$ : Total number of segments composing a message to be sent from processor  $i$  in the compact message scheme, where  $0 \leq Gs_i \leq E_i$  and  $0 \leq i < P$ .
- $Gr_i$ : Total number of segments in a received message on processor  $i$  in the compact message scheme, where  $0 \leq Gr_i \leq E_a$  and  $0 \leq i < P$ .
- $\delta_i$ :  $\delta_i L$  elements in the local mask array located on processor  $P_i$  have true values. That is,  $E_i = \delta_i L$ .

On processor  $i$  ( $0 \leq i < P$ ) the local computation time would be proportional to

$$\alpha L + \beta C + \gamma E_i + \eta E_a + \epsilon Gs_i + \zeta Gr_i,$$

where the values of  $\alpha, \beta, \gamma, \eta, \epsilon$ , and  $\zeta$  depend on the scheme.

For simplicity in modeling we consider only a one-dimensional input array. Also, we assume that the *true* values are distributed randomly in an input mask array. Since the intermediate step in the ranking stage is common for all three schemes and the detailed analysis has already been presented in Section 5, we will focus on the storage scheme and the message composition scheme, which are the initial and the final steps in the ranking stage, and the message composition/decomposition step in the redistribution stage. Given a one-dimensional input array, the local computation for the intermediate step takes time  $\Theta(C)$ , which is common for all three schemes.

#### 6.4.1 Storage Scheme

The simple storage scheme requires only one local scan in the initial step, but it requires maintaining information for each local packed element. The information consists of  $d + 3$  items: a local index on each dimension, a tile number, a rank, and a destination processor number. For each item we need at least one memory read and write operation. Thus maintaining information for local packed elements will take time  $\Theta(4E_i)$ .

A communication message consists of the pair (*datum value, global rank*), so the size of the message to be sent from processor  $i$  is  $2E_i$  and the size of the received message is  $2E_a$ . Therefore, the message composition requires time  $\Theta(2E_i)$  and the message decomposition requires time  $\Theta(2E_a)$ . The local computation time will be proportional to  $L + C + 6E_i + 2E_a$ .

The compact storage scheme requires two local scans, one in the initial step and the other in the message composition step. Also, in the final step we make the *sendl* vector. Suppose slice  $s_j$  ( $0 \leq j < C$ ) consists of  $k_j$  ( $0 \leq k_j \leq P$ ) segments on processor  $i$ . Then we need to check  $C$  slices, and for each slice  $s_j$  we are required to access *sendl* vector  $k_j$  times. Since  $Gs_i = \sum_{j=0}^{C-1} k_j \leq E_i$ , the time for making a *sendl* vector is upper-bounded by  $C + E_i$ . Note that the structure of the message is the same as that in the simple storage scheme. Therefore, the local computation time will be proportional to  $2L + 2C + 3E_i + 2E_a$ .

**Comparison** The local computation times for the compact storage scheme will be less than for the simple storage scheme when

$$L + C = L + \frac{L}{W_0} \leq 3E_i = 3\delta_i L.$$

The main factors characterizing the performances of the two schemes will be *block size*,  $W_0$ , and *mask density*,  $\delta_i$ . With relative low mask density,  $\delta_i$ , we can expect the simple storage scheme to work better than the compact storage scheme. Also, as the distribution is close to cyclic distribution (that is,  $W_0$  is close to 0), we can expect the simple storage scheme to give the better performance. Actually, in cyclic distribution,  $C = L$ . It follows that the compact storage scheme will give the worst performance in cyclic distribution, but, as the block size,  $W_0$ , is close to the local array size,  $L$  (that is, the input distribution is close to block distribution), the compact storage scheme will work better. Since  $C = 1$  in block distribution, the compact storage scheme works best in block distribution.

#### 6.4.2 Message Composition/Decomposition Scheme

In the compact message scheme a message consists of segments, with each segment having the form (*base-rank*, *number of data in the segment*, *datum*,  $\dots$ , *datum*), so that the size of the message to be sent from processor  $i$  is  $E_i + 2Gs_i$ , and the size of the received message is  $E_a + 2Gr_i$ . Also, note that the compact message scheme basically uses the same storage scheme as that used in the compact storage scheme. Therefore, the local computation time of the compact message scheme will be proportional to  $2L + 2C + 2E_i + 2Gs_i + E_a + 2Gr_i$ , where  $0 \leq Gs_i \leq E_i$  and  $0 \leq Gr_i \leq E_a$ .

**Comparison** The local computation times for the compact message scheme will be less than for the compact storage scheme when

$$2(Gs_i + Gr_i) \leq E_i + E_a.$$

If  $Gs_i < E_i/2$ , the size of the message sent from processor  $i$  in the compact message scheme is smaller than that in the compact storage scheme. Similarly, if  $Gr_i < E_a/2$ , the size of the message received from other processors in the compact message scheme is smaller than that in the compact storage scheme. As mentioned before, in cyclic distribution the compact storage scheme always has a smaller message. In block distribution (i.e.,  $C = 1$ ),  $Gs_i \leq P$  and the average number of elements in each segment will be equal to or larger than  $E_i/P$ . Therefore, the compact message scheme will work best in block distribution. Generally, the total number of segments,  $Gs_i + Gr_i$ , depends on the block size,  $W_0$ , and the mask density,  $\delta_i$ .<sup>6</sup> Given a fixed mask density, as  $W_0$  increases,  $Gs_i + Gr_i$  will decrease, so that the compact message scheme will show a better performance. Also, given a fixed  $W_0$ , as the mask density increases, the total number of elements in each segment will increase, so that the compact message scheme will give a better performance.

---

<sup>6</sup>As mentioned before, if the result vector is not distributed in block,  $Gs_i + Gr_i$  will increase as the block size of the result vector decreases.

## 7 Experimental Results

Three schemes (*simple storage scheme*, *compact storage scheme* and *compact message scheme*) for pack, and two schemes (*simple storage scheme* and *compact storage scheme*) for UNPACK, were programmed in C on the CM-5. Experiments were conducted for six one-dimensional arrays ( $N = 4096, 8192, 16384, 32768, 65536, 131072$ ) and four two-dimensional arrays ( $N \times N = 64 \times 64, 128 \times 128, 256 \times 256, 512 \times 512$ ). On the CM-5, 16 processors for one-dimensional arrays and  $4 \times 4$  processors for two-dimensional arrays were used.

In addition, other experiments were performed on the CM-5 by using 256 processors ( $16 \times 16$  for a two-dimensional array), a number 16 times greater than the 16 processors used in the former experiments, in the following manner. We first chose two input arrays,  $N = 65536$  and  $N \times N = 512 \times 512$ , and then increased the size of the input arrays 16 times as we increased the number of processors 16 times. Hence in two experiments the local array size was fixed, but the number of processors was increased 16 times.

Various block sizes were used in our experiments for the distribution of those arrays, but the block size for dimension 0 was fixed to be the same as that for dimension 1 in the two-dimensional arrays in order to study the sensitivity of the performance to the block size.

The result vector  $V$  in PACK (the input vector  $V$  in UNPACK) was fixed to be distributed in block. Five input mask arrays were randomly generated with density = 10%, 30%, 50%, 70%, and 90%, and one mask array was made in such a way that the mask value was true in the one-dimensional array if the global index was less than  $N/2$ , and that in the two-dimensional array was true if the global index on dimension 1 was larger than that on dimension 0. Here the mask density  $\delta$  means that the  $\delta\%$  of elements in the mask array have true values. We measured time for local computation in PACK, time for many-to-many personalized communication in PACK, total time for PACK/UNPACK, and time for the prefix-reduction-sum in PACK/UNPACK. Also, we compared the performances of three different schemes for PACK (two schemes for UNPACK). But note that only partial results are presented in Figures 3 through 5 due to space limitation. Full results are available in [1].

### Local Computation

The parallel PACK/UNPACK algorithms consist of two stages, *ranking* and *redistribution*, and the local computation time measured in our experiment includes the time for the ranking stage (excluding the time taken by the prefix-reduction-sum) and the time for message composition and decomposition in the redistribution stage. The execution time for the local computation in PACK is shown in Figure 3. The local computation time increases as the block size decreases, independent of the mask density. The performance of the intermediate step and computing the final base-rank array in the final step of the ranking stage are proportional to the number of tiles.

Assume that the compact storage scheme and the compact message scheme are better than the

simple storage scheme when the block size is equal to or greater than  $\beta_1$  and  $\beta_2$ . Both  $\beta_1$  and  $\beta_2$  are always greater than 1. Thus for cyclic distribution the simple storage scheme is always better than the other two schemes. As the block size increases, two optimized schemes give a much better performance than the simple storage scheme does, thus we have the best results in the block distribution.

Table I presents the  $\beta_1$  values for different mask densities. We can see that  $\beta_1$  is very large for a mask density of 10%. The amount of memory access in the simple storage scheme largely depends on the mask density, thus the values of  $\beta_1$  decrease as the mask density increases. When the block size is greater than  $\beta_1$ , the compact message scheme is better than the compact storage scheme.

Table I:  $\beta_1$  values for 6 different mask densities

<i>One-dimensional input array</i>							<i>Two-dimensional input array</i>						
Local Size	10%	30%	50%	70%	90%	LT	Local Size	10%	30%	50%	70%	90%	LT
1024	64	8	8	4	4	4	16	$\infty$	4	4	2	2	2
2048	128	16	8	4	4	4	32	$\infty$	8	2	2	2	2
4096	512	16	8	4	4	4	64	32	8	2	2	2	2
8192	2048	8	8	4	4	4	128	16	4	4	2	2	2

## Many-to-Many Personalized Communication

In our experiments the result vector in PACK (the input vector in UNPACK) was fixed to be distributed in block, and the *linear permutation scheduling* algorithm [9] using *active messages* [4] was used for many-to-many personalized communication. Readers are referred to [1] for a comparison of different communication scheduling algorithms.

Results for the communication step are similar to those for the local computation. That is, as mask density increases the compact message scheme gives a better performance than the simple storage scheme. Generally, as the block size increases, the compact message scheme gives a better performance than the simple storage scheme except in block distribution (especially, for a one-dimensional input array). When an input array is distributed in block, each processor will send most parts of the message to itself, because we assume the elements to be packed are randomly distributed in the input array and the result vector is distributed in block. If the elements to be packed are not randomly distributed, that will not happen. Also, note that in our implementation local copy was not performed when a processor needed to send a message to itself.

## Vector Prefix-Reduction-Sum

In our implementation CM-5 global communication functions were used for one-dimensional input arrays. For two-dimensional input arrays, the direct algorithm was used if the number of processors was less than or equal to 4 or the vector size was less than the number of processors. Otherwise, the split algorithm was used.

The performance of prefix-reduction-sum depends completely on the size of the vector on which it is performed. Hence, as the block size increases the time taken by prefix-reduction-sum decreases. Also, the time for prefix-reduction-sum increases more rapidly in two-dimensional arrays than in one-dimensional arrays as the block size decreases. For extensive results, including a comparison of direct algorithm and split algorithm, readers are referred to [6].

## Total Execution Time in PACK/UNPACK

Figures 4 and 5 show the total execution times in PACK/UNPACK. For a fixed local array size, the total costs for PACK/UNPACK are dominated by the cost for local computation in a small number of processors. But in a large number of processors the most time is spent for communication (vector prefix-reduction-sum and many-to-many personalized communication). Generally, the time for the vector prefix-reduction-sum is relatively small when compared to that for local computation and for many-to-many personalized communication. Only when the block size is very small (especially, block size= 1), is the time for the vector prefix-reduction-sum larger than that for many-to-many personalized communication, but smaller than that for local computation.

As can be seen in the results for local computation and many-to-many personalized communication, the compact message scheme gives the best performance of three schemes, and the compact storage scheme gives better results than the simple storage scheme when block size is relatively large and mask density is relatively high.

## Redistribution Scheme

Table II shows the execution time for two redistribution schemes, *redistribution of selected data* and *redistribution of whole arrays*, and the simple storage scheme, with the input array distributed cyclicly. In Table II the timing was calculated in such a way that the time for preliminary redistribution was added to the total time for the compact message scheme with the input array distributed in block. Note that the simple storage scheme gives the best results when the input array is distributed cyclicly, but the compact message scheme gives the best results when the input array is distributed in block.

In one-dimensional arrays the total costs for redistribution are dominated by the cost for communication detection. Therefore, as mentioned before, the second redistribution scheme gives poorer results. Neither of the two redistribution schemes gives better results than the simple storage scheme. In two-dimensional arrays the first redistribution scheme (redistribution of selected data) is better than the

simple storage scheme for relatively low mask densities, while the second redistribution scheme (redistribution of whole arrays) is better for relatively high mask densities. The performance of the second redistribution scheme is not greatly affected by mask density.

Table II: Comparison of execution time (*msec*) for two Redistribution schemes in parallel PACK (Number of processors= 16 for 1-D arrays,  $4 \times 4$  for 2-D arrays)

Mask	16384			65536			$256 \times 256$			$512 \times 512$		
Density	SSS	Red. 1	Red. 2	SSS	Red. 1	Red. 2	SSS	Red. 1	Red. 2	SSS	Red. 1	Red. 2
10%	8.83	139.70	382.13	30.52	553.03	1520.89	46.44	34.83	60.62	184.49	110.86	195.85
30%	10.89	141.80	382.51	37.31	558.36	1520.23	56.23	49.57	62.31	219.97	170.63	202.04
50%	12.40	143.29	382.67	44.58	565.24	1521.53	64.76	63.99	64.24	256.37	231.35	208.66
70%	14.09	144.86	382.94	51.33	571.75	1522.37	73.58	78.16	65.64	293.51	288.22	214.74
90%	15.66	146.63	383.25	58.64	578.38	1523.36	82.38	91.59	66.97	327.34	345.72	220.59

## 8 Conclusions

We have presented a parallel PACK/UNPACK algorithm using a ranking algorithm and have shown the experimental results for PACK/UNPACK on the CM-5. The ranking algorithm computes the rank of each element to be packed using parallel vector prefix-sum and reduction-sum operations without actually moving any elements among processors. We have also presented two optimization schemes for parallel PACK/UNPACK, *compact storage scheme* and *compact message scheme*. For the cyclicly distributed input array we have suggested two redistribution schemes, *redistribution of selected data* and *redistribution of whole arrays*.

The performance of the ranking algorithm largely depends on the block size of input arrays distributed in block-cyclic, especially the block size of the lower dimension. Hence we have the lowest ranking overhead when the input array is distributed in block. When the input array is distributed cyclicly, two redistribution schemes may be used to reduce the ranking overhead. The performance of the ranking algorithm may not be greatly affected by the total number or by the distribution of the elements to be packed.

The compact message scheme gave the best performance among three schemes, and the compact storage scheme gave better results than the simple storage scheme when the block size was relatively large and the mask density relatively high. This was mainly due to the reduction of local computation in the compact storage scheme and the reduction of communication overhead in the compact message scheme.

## Acknowledgments

We are grateful to Northeast Parallel Architectures Center and Minnesota Supercomputing Center for allowing us to use their CM-5. Also, we would like to thank Elaine Weinman for proofreading this paper. The work of Sanjay Ranka was supported in part by NSF under ASC-9213821 and AFMC, and by ARPA under contract #F19628-94-C-0057. The content of the information does not necessarily reflect the position or the policy of the United States government and no official endorsement should be inferred.

## References

- [1] Seungjo Bae and Sanjay Ranka. Parallel Ranking Algorithm for PACK/UNPACK on the Coarse-Grained Distributed Memory Parallel Machines. Technical report, School of Computer and Information Science, Syracuse University, September 1995.
- [2] W. S. Brainerd, C. H. Goldberg, and J. C. Adams. *Programmer's Guide to FORTRAN 90*. McGraw Hill, 1990.
- [3] Thinking Machines Corporation. *CM Fortran Language Reference Manual: Version 2.1*.
- [4] T. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. Active Messages: a mechanism for integrated communication and computation. In *Proc. of ISCA 1992*, pages 256–266, 1992.
- [5] High Performance Fortran Forum. *High Performance Fortran Language Specification: Version 1.0*, May 1993.
- [6] Dongmin Kim, Seungjo Bae, and Sanjay Ranka. Split Algorithm for Parallel Vector Prefix and Reduction. Technical report, School of Computer and Information Science, Syracuse University, July 1995.
- [7] S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines on distributed memory multicomputers. In *Proc. of Frontiers' 95*, 1995.
- [8] Ravi V. Shankar and Sanjay Ranka. Random Data Accesses on a Coarse-grained Parallel Machine I. One-to-One Mappings. Submitted to *Journal of Parallel and Distributed Computing*, 1995.
- [9] J-C. Wang, T-H. Lin, and S. Ranka. Distributed Scheduling of Unstructured Collective Communication on the CM-5. In *Proc. of Hawaii International Conference on System Sciences*, 1993.

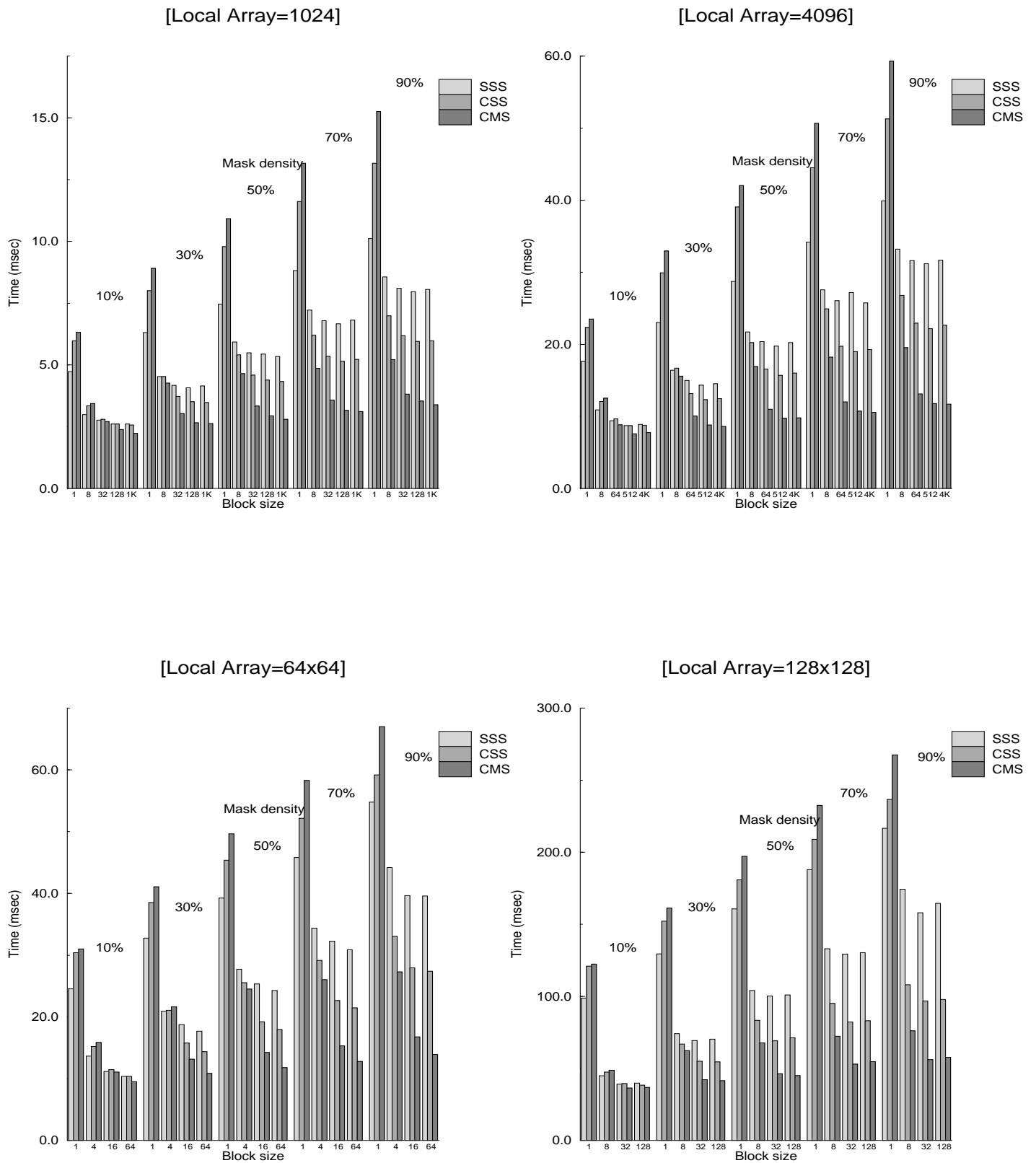


Figure 3: Comparison of local computation time (*msec*) for three schemes in PACK (SSS: Simple Storage Scheme, CSS: Compact Storage Scheme, CMS: Compact Message Scheme)

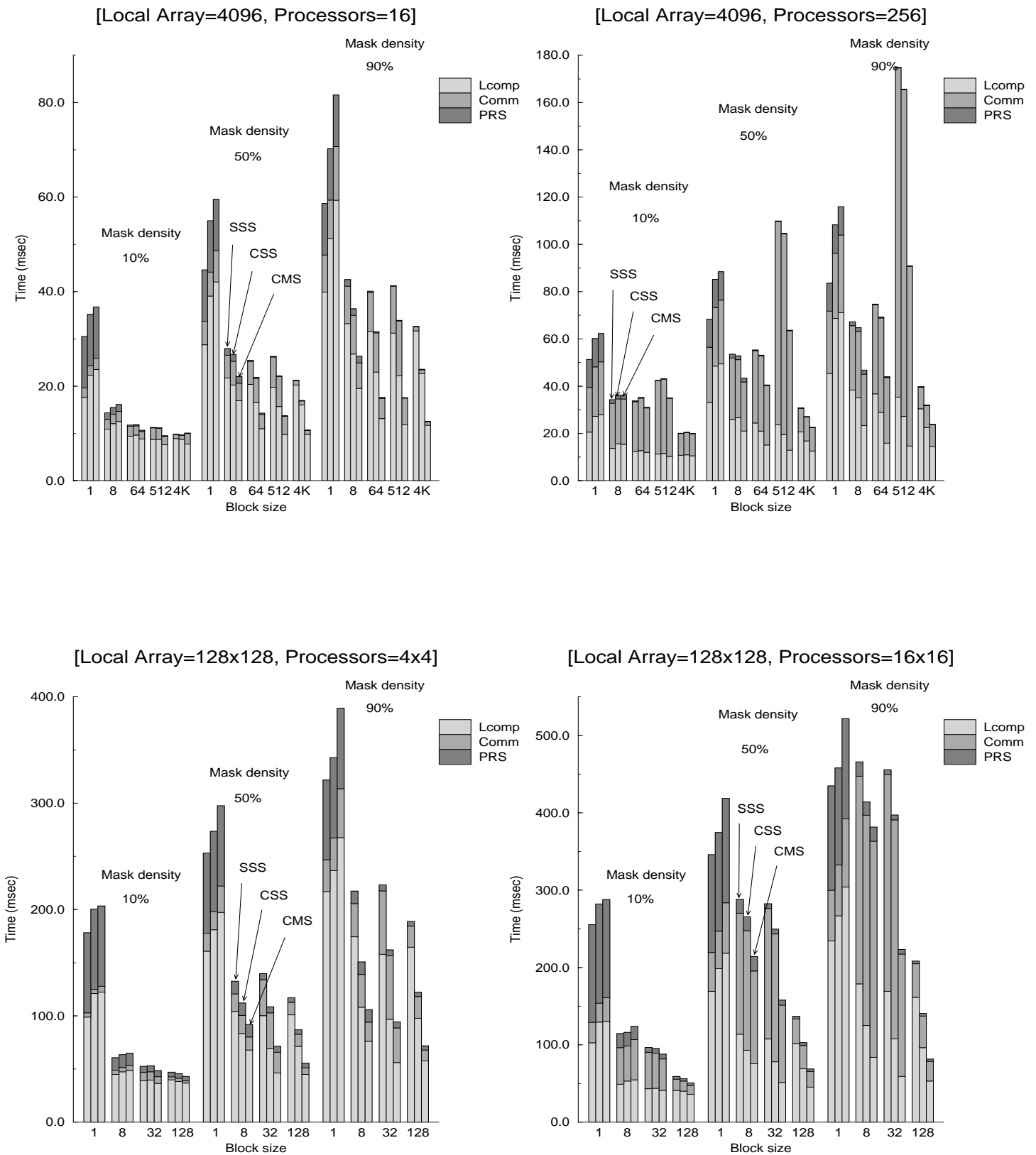


Figure 4: Comparison of total execution time (*msec*) for three schemes in PACK (SSS: Simple Storage Scheme, CSS: Compact Storage Scheme, CMS: Compact Message Scheme)

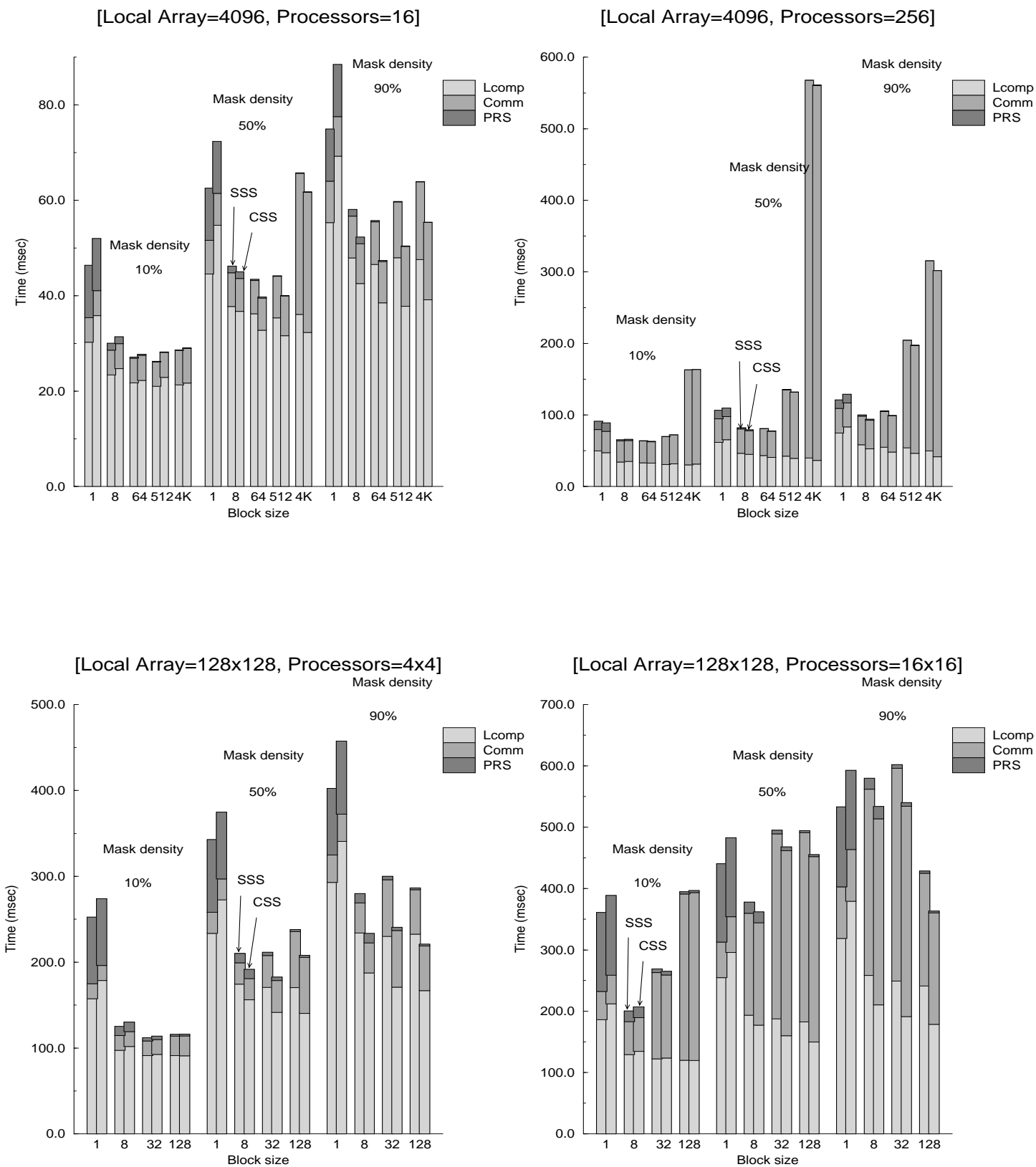


Figure 5: Comparison of total execution time (*msec*) for two schemes in UNPACK (SSS: Simple Storage Scheme, CSS: Compact Storage Scheme)