PACK/UNPACK on Coarse-Grained Distributed Memory Parallel Machines

Seungjo Bae
2–120 Center for Science and Technology
School of Computer and Information Science
Syracuse University, Syracuse, NY 13244-4100

Sanjay Ranka
CSE Building, #301
Department of Computer Science
University of Florida, Gainesville, FL 32611

Abstract

PACK/UNPACK are Fortran 90/HPF array construction functions which 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.

1. Introduction

PACK/UNPACK are Fortran 90/HPF array construction functions which derive new arrays from existing arrays [2, 4]. PACK gathers selected elements from an input array of any rank under the control of a logical mask array and constructs the new vector. 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 elements of A (or M in UNPACK) under the control of M.

2. *Redistribution*: Redistribute all corresponding elements of A (or V) to V (or A).

In this paper we present 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 definitions and basic operations are presented in Section 2. In Section 3 we describe the parallel PACK/UNPACK algorithm using the parallel ranking algorithm, and we present the parallel ranking algorithm in Section 4. We explain the optimized schemes in Section 5. Section 6 presents the experimental results, and we state our conclusions in Section 7.

2. Definitions and Basic Operation

Definitions and Notations PACK gathers elements of input array A to vector V under the control of mask array M, which should be conformable with A. We assume that M is aligned to A. The size of V is the same as the number of $true$ values in M. All notations used in this paper are as follows:

A	Array of rank d and shape $(N_{d-1}, \ldots, N_0)$
A_l	Local array of rank d and shape $(L_{d-1}, \ldots, L_0)$
M	Mask array of rank d and shape $(N_{d-1}, \ldots, N_0)$
V	Vector (rank-one array)
Pn	Processor number
N	Global array size ($N = N_{d-1} \times \cdots \times N_0$)
P	Number of processors ($P = P_{d-1} \times \cdots \times P_0$)
L	Local array size ($L = N/P = L_{d-1} \times \cdots \times L_0$)
N_i	Global array size on a dimension i
P_i	Number of processors on a dimension i
W_i	Block size on a dimension i in block-cyclic distribution
L_i	Local array size on a dimension i ($L_i = N_i/P_i$)
S_i	Tile size on a dimension i ($S_i = P_i\, W_i$)
T_i	Total number of tiles on a dimension i ($T_i = N_i/S_i$)
$Size$	Total number of $true$ values in mask array M
Δ	Sub-domain where the intermediate ranks are valid

For the sake of simplicity, we assume that for all i $(0 \leq i < d)$, $P_i \backslash N_i$ and $W_i \backslash N_i$. Also, we assume that the row-majored indexing scheme is used and all indices start from zero. The above definitions and notations presented for PACK are also valid for UNPACK, except that UNPACK scatters elements of input vector V of size N' to A under the control of M. Note that N' should not be less than $Size$.

Vector prefix-reduction-sum The *vector prefix-reduction-sum* primitive, which is used in the parallel ranking algorithm, performs the element-wise vector prefix-sum and vector reduction-sum simultaneously where the prefix operation is exclusive [5]. We also define the cost function for vector prefix-reduction-sum as $\mathsf{PRS}(P, M)$ where P is the number of processors and M is the vector size. Then, $\mathsf{PRS}(P, M)$ is $\Omega(\tau \lg P + \mu M)$ without a control network, and $\Theta(M)$ with a control network as on the CM-5 where τ is the startup time and μ is the per-element transfer time. For details of algorithms, readers are referred to [5].

3. 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 algorithm, which will be presented in Section 4.

3.1. 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.

3.2. UNPACK

For all i $(0 \le i < Size)$, $V[i]$ should be transferred to the element of A whose corresponding element of M has the rank i 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.

4. Parallel ranking algorithm

The ranking algorithm computes the rank of each element to be packed using the parallel *vector prefix-reduction-sum* primitive without actually moving any elements among processors. The ranking algorithm consists of three steps: *initial step*, *intermediate step*, and *final step*. For each dimension i $(0 \le i < d)$, we use the two local offset arrays, PS_i (for exclusive prefix-sum) and RS_i (for reduction-sum), of rank $d - i$ and shape $(L_{d-1}, \ldots, L_{i+1}, T_i)$ in order to compute the intermediate rank. We thus need a total of $2d$ local arrays. The final rank is derived by summing all the offset arrays.

The performance of the ranking algorithm mainly depends on the value of T_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. The rest of the presentation is limited to PACK for obvious reasons.

Initial step (local scan) We scan all local elements of A_l to find the elements to be packed and to set the initial rank of those elements under the control of M, initializing PS_0 and RS_0. At the end of this step, $PS_0(:, \cdots, :, i)$ and $RS_0(:, \cdots, :, i)$ are the total number of local packed elements in $A_l(:, \cdots, :, i\,W_0 : (i+1)\,W_0 - 1)$ where $0 \le i < T_0$, and $\Delta = [1 \times \cdots \times 1 \times W_0]$. This initial step takes time $\Theta(L)$.

Intermediate step The intermediate step consists of d (rank of the input array) steps. Each intermediate step i $(0 \le i < d)$ consists of three sub-steps:

- **Input from the intermediate Step $i - 1$**
  Initialized PS_i and $RS_i(L_{d-1}, \cdots, L_{i+1}, T_i)$ where $\Delta = [1 \times \cdots \times 1 \times W_i \times N_{i-1} \times \cdots \times N_0]$.
- **Substep 1: Vector prefix-reduction-sum**
  Vector prefix-reduction-sum on PS_i (exclusive prefix sum) and RS_i (reduction sum) along the dimension i where $\Delta = [1 \times \cdots \times 1 \times S_i \times N_{i-1} \times \cdots \times N_0]$. This step takes time $\Theta(\,\mathsf{PRS}(P_i,\ (\prod_{k=i+1}^{d-1} L_k) T_i)\,)$.
- **Substep 2: Local prefix sum**
  1. $RS_{i+1}(:, \cdots, :, k) \leftarrow RS_i(:, \cdots, :, l, T_i - 1)$ where $0 \le k < T_{i+1}$ and $l = (k+1)W_{i+1} - 1$. In step $d-1$, $Size \leftarrow RS_{d-1}(T_{d-1} - 1)$.
  2. Set the starting point of each segment (of total T_{i+1} segments) at $RS_i(:, \cdots, :, m, 0)$ for all m such that $m \bmod W_{i+1} = 0$.
  3. Segmented exclusive prefix-sum on RS_i using the same indexing scheme.
  4. $PS_i \leftarrow PS_i + RS_i$.

  Now Δ is $[1 \times \cdots \times 1 \times W_{i+1} \times N_i \times N_{i-1} \times \cdots \times N_0]$, and this step requires time $\Theta(\,(\prod_{k=i+1}^{d-1} L_k) T_i\,)$.
- **Substep 3: Modify PS_{i+1} and RS_{i+1}**
  PS_{i+1} and $RS_{i+1}(:, \cdots, :, k) \leftarrow RS_{i+1}(:, \cdots, :, k) + RS_i(:, \cdots, :, l, T_i - 1)$ where $0 \le k < T_{i+1}$ and $l = (k+1)W_{i+1} - 1$. In step $d-1$, $Size \leftarrow Size + RS_{d-1}(T_{d-1} - 1)$. This step takes time $\Theta(\,(\prod_{k=i+2}^{d-1} L_k) T_{i+1}\,)$

Final step We compute the global rank of all packed elements by summing all PS_i $(0 \le i < d)$. Now the subdomain Δ is the same as 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 α is the maximum number of local packed elements among all processors $(0 < \alpha \le L)$.

5. Optimization

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. First of all, we define *slice*, which is a portion of a local array, A_l, such as $A_l(:, \cdots, :, i\,W_0 : (i+1)\,W_0 - 1)$ where $0 \le i < T_0$.

5.1. Compact storage scheme

We first present *simple storage scheme* and then explain the optimized scheme. In the simple storage scheme information regarding local elements to be packed are saved in the initial step, and then used in the final step of the ranking stage. This information includes a local index on each dimension, a tile number and an initial local rank. Therefore, this scheme requires at least four memory-write and memory-read operations for each local element to be packed. 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, we copy PS_0 to the counter array, PS_c before the intermediate step. After computing the final base-rank array, PS_f, we can derive the size of the message, which will be sent to each destination processor, from comparing each element of PS_c with that of PS_f during the final step. Hence, the time taken by the final step depends mainly on the number of tiles on dimension 0, T_0, since the size of two arrays is $\left(\prod_{i=1}^{d-1} L_i\right) T_0$. In the redistribution stage, message-coalescing requires another local scan of the input array. However, this local scan can be done more efficiently than that in the initial step, since we know the total number of local packed elements in each slice. This may be especially useful when both the slice size and the total number of packed elements are relatively large.

5.2. Compact message scheme

In the redistribution stage of PACK, we need to send the selected datum value together with its global rank. In the simple storage scheme the communication message consists of the pair (*datum value, global rank*), so that a message is twice as long as the number of packed elements.

To reduce communication overhead (message size), we alternatively use the *compact message scheme*. Suppose there are n packed elements in a slice. Then, we can partition these n values into i segments ($0 \leq i < P$) according to their destination processor. 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. Therefore, the communication message consists of segments, with each segment having the form of (*rank of the first packed element in the segment, number of packed elements in the segment, datum, . . ., datum*).

Since the size of each segment is at least 3, this optimized scheme may not be useful if the slice size is one or if there is only one packed element in a slice. The performance of this compact message scheme depends on the block size on dimension 0 as well as on the number of packed elements in each slice. Also, the performance depends on the block size of the result vector.

5.3. Cyclic to block distribution

Suppose a 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 the dimensions. Once the distribution is done as the preliminary step we can apply the ranking algorithm with the lowest ranking overhead, since all dimensions are distributed in block.[1] The details of the communication detection algorithms for array redistribution can be found in [6]. In the following, we describe two redistribution schemes as the preliminary steps in PACK.

In the first scheme (*redistribution of selected data*) we redistribute selected input data to be packed, but no data from the mask array. 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. On the other hand, when the total number of the packed elements is relatively large, the second scheme (*redistribution of whole array*) is more useful. That is, we redistribute the input array and the mask array in order to reduce the local computation overhead entailed in the previous redistribution scheme.

6. 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 4×4 processors for two-dimensional arrays were used.

In addition, other experiments were performed on the CM-5 by using 256 processors (16×16 for a two-dimensional array), which is 16 times larger 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, but W_0 was fixed to be the same as W_1 in the two-dimensional arrays. The vector V was fixed to be distributed in block. Five input mask arrays were randomly generated with density $= 10\%, 30\%, 50\%, 70\%$, and 90%,.[2] Notice that only partial results are presented in Tables 1 and 2 due to space limitation. Full results are available in [1].

[1] Note that this distribution method can not be used in UNPACK, since UNPACK is a *READ (or GATHER)* operation, so that the input data should be maintained in the original distribution.

[2] Here the mask density δ means that the $\delta\%$ of elements in the mask array have true values.

Array size = 65536, Number of processors = 16

Block size	Mask density = 10%			Mask density = 50%			Mask density = 90%		
	SSS	CSS	CMS	SSS	CSS	CMS	SSS	CSS	CMS
1	30.52	35.19	36.77	44.58	54.99	59.55	58.64	70.22	81.59
8	14.38	15.48	16.10	27.96	26.69	22.05	42.55	36.39	26.35
64	11.77	11.85	10.68	25.45	21.87	14.28	40.07	31.52	17.65
512	11.25	11.19	9.50	26.28	22.15	13.73	41.23	33.89	17.58
4096	9.84	9.67	10.05	21.25	16.96	10.69	32.64	23.55	12.52

Array size = 512 $\times$ 512, Number of processors =4 $\times$ 4

Block size on dim 0, 1	Mask density = 10%			Mask density = 50%			Mask density = 90%		
	SSS	CSS	CMS	SSS	CSS	CMS	SSS	CSS	CMS
1	184.49	205.93	208.21	256.37	279.49	303.09	327.34	351.71	398.42
8	61.00	62.46	64.50	133.40	109.23	91.24	204.07	148.87	105.41
32	52.99	52.02	48.69	142.49	111.07	73.18	236.66	167.02	94.52
128	47.60	48.98	42.25	121.95	88.03	55.06	188.89	126.93	66.99

Table 1. Comparison of total execution time $(millisec)$ for three schemes in PACK (SSS: Simple Storage Scheme, CSS: Compact Storage Scheme, CMS: Compact Message Scheme)

Local computation: As the block size, W_i $(i = 0, 1)$, decreases, local computation time increases no matter how high the mask density is. The performance of the intermediate step and computing the final base-rank array, PS_f, in the final step of the ranking stage is proportional to the number of tiles, T_i, where $T_i = N_i/(P_i W_i)$ $(i = 0, 1)$. In the intermediate step the size of two arrays, PS_i and RS_i, increases as the block size decreases (or as the number of tiles increases). Similarly, the time for computing the final base-rank array, PS_f, during the final step increases as the block size decreases. As the mask density increases, the compact message scheme and the compact storage scheme give a much better performance than the simple storage scheme. Since the compact message scheme basically requires the compact storage scheme, the compact message scheme gives a better performance than the compact storage scheme except when the block size is 1 or 2. In cyclic distribution we had the worst performance. Also, the simple storage scheme gave the better results than two optimized schemes.

Many-to-many personalized communication: For many-to-many personalized communication, *linear permutation scheduling* algorithm [7] using *active messages* [3] was used. We have results similar to those for 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.

Vector prefix-reduction-sum: The performance of *vector prefix-reduction-sum* depends completely on the size of the vector on which prefix-reduction-sum is performed. The vector size at the intermediate step i $(0 \leq i < d)$ is $\left(\prod_{k=i+1}^{d-1} L_k\right) T_i$. In one-dimensional arrays, the vector size is $T_0 = L_0/W_0$. In two-dimensional arrays, the vector sizes at the intermediate steps 0 and 1 are $L_1 T_0 = (L_1 L_0)/W_0$ and $T_1 = L_1/W_1$, respectively. Hence, as the block size increases (or the total number of tiles decreases), the time taken by vector prefix-reduction-sum decreases.

Total execution time 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, 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), the time for the vector prefix-reduction-sum is 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 among three schemes, and the compact storage scheme gives better results than the simple storage scheme when the block size is relatively large and the mask density is relatively high.

Redistribution scheme: In one-dimensional arrays a preliminary redistribution requires much more time for communication detection than in two-dimensional arrays. Neither of the two redistribution schemes gives better results than the simple storage scheme. In two-dimensional arrays the first redistribution scheme gives better results when the mask density is relatively low (Table 3). The performance of the second redistribution scheme is not greatly affected by the mask density, but the performance of the first redistribution scheme depends on the mask density.

7. Conclusions

We have presented a parallel PACK/UNPACK algorithm using the ranking algorithm and have shown the experimental results for PACK/UNPACK on the CM-5. We have also presented two optimization schemes, *compact storage*

Array size = 65536, Number of processors = 16

Block size	Mask density = 10%		Mask density = 50%		Mask density = 90%	
	SSS	CSS	SSS	CSS	SSS	CSS
1	46.38	52.02	62.56	72.38	74.96	88.49
8	30.02	31.38	46.22	45.01	58.09	52.30
64	27.14	27.73	43.43	39.74	55.73	47.39
512	26.23	28.22	44.17	40.06	59.76	55.42
4096	28.63	29.05	65.76	61.77	63.94	55.43

Array size = 512 $\times$ 512, Number of processors = 4 $\times$ 4

Block size on dim 0, 1	Mask density = 10%		Mask density = 50%		Mask density = 90%	
	SSS	CSS	SSS	CSS	SSS	CSS
1	252.57	274.01	342.81	374.73	402.47	457.44
8	125.32	130.25	210.15	191.78	279.88	233.60
32	112.05	113.92	211.66	182.67	300.00	240.72
128	116.10	116.21	237.99	207.94	286.59	221.07

Table 2. Comparison of total execution time $(millisec)$ for two schemes in UNPACK (SSS: simple storage scheme, CSS: compact storage scheme)

Mask Density	256 $\times$ 256			512 $\times$ 512		
	SSS	Red. 1	Red. 2	SSS	Red. 1	Red. 2
10%	46.44	34.83	60.62	184.49	110.86	195.85
50%	64.76	63.99	64.24	256.37	231.35	208.66
90%	82.38	91.59	66.97	327.34	345.72	220.59

Table 3. Comparison of execution time $(millisec)$ for two redistribution schemes in PACK (number of processors $= 4 \times 4$)

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 on 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 the distribution of the elements to be packed. Parallel PACK/UNPACK using the ranking algorithm are architecture independent and can be applied to arrays of arbitrary dimensions with arbitrary distribution along every dimension.

The compact message scheme has given the best performance of three schemes, and the compact storage scheme has given better results than those of the simple storage scheme when the block size is relatively large and the mask density is relatively high. Also, as the local array size increases, two optimized schemes gave a much better performance. This is mainly due to the reduction of local computation in the compact storage scheme and the reduction of the communication overhead in the compact message scheme.

References

[1] Seungjo Bae and Sanjay Ranka. Parallel Ranking Algorithm for PACK/UNPACK on 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] T. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proc. of ISCA 1992*, pages 256–266, 1992.

[4] C. H. Koelbel et. al. *The High Performance Fortran Handbook*. MIT Press, 1994.

[5] 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.

[6] S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines on distributed memory multicomputers. In *Proc. of Frontiers' 95*, 1995.

[7] 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.