HPF Parallel Prefix / Suffix Intrinsic Functions

Yuhong Wen, Bryan Carpenter, Geoffrey C. Fox

Northeast Parallel Architecture Center
at Syracuse University
111 College Place
Syracuse, NY, 13244-4100
{wen, dbc, gcf}@npac.syr.edu

Abstract

Parallel prefix and suffix functions are very important intrinsic functions in HPF (High Performance Fortran) language’s runtime-system libraries. Several algorithms have been published to deal with this parallel prefix / suffix problem. But there are some specifications should be considered in HPF to implement this kind of functions with high performance. In HPF, there are different kinds of array data distribution models, such as BLOCK, CYCLIC, BLOCK-CYCLIC, which makes it difficult to implement these functions efficiently, especially for BLOCK-CYCLIC case. In this paper, we will survey all these kinds of data distribution models, and introduce a new algorithm for every distribution model to implement parallel prefix and suffix functions with high performance on distributed memory systems based on message passing communication. From the algorithm we can see, we can deal with the difficult BLOCK-CYCLIC distribution as efficiently as the BLOCK and CYCLIC distributions.

Keywords:
Data distribution model, Scan operation, Parallel Prefix / Suffix intrinsic functions, High Performance Fortran (HPF)

1. Introduction

Array prefix and suffix functions are very important group of operations in HPF intrinsic functions. Any prefix
and suffix function will scan the whole vector to associate with a kind of binary operation, each element of the result is a function of the elements of the vector that precede it (for a prefix scan) or that follow it (for a suffix scan). These functions provide scan operations on arrays and subarrays. These functions have the following general form: [1]

XXX_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
XXX_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

The allowed values of XXX are ALL, ANY, COPY, COUNT, IALL, IANY, IPARITY, MAXVAL, MINVAL, PARITY, PRODUCT, and SUM.

Generally, prefix and suffix operations have been widely used in scientific computing and this kind of problem has been around for centuries as the recurrence computing of \( x_i = a_i + x_{i-1} \). [2] To implement this operation in parallel, Ladner and Fischer first showed an efficient general-purpose circuit for implementing the scan operation [3], and the work of Fich [4], and Lakshminarahan, Yang and Dhall [5] gave improvements over the circuit of Ladner and Fischer. In the CRCW PRAM model one can do even better, in that, as shown by Cole and Vishkin[8], one can achieve a running time of \( O(\log n/\log \log n) \) using an optimal number of processors. This time was shown to be the best possible with any polynomial number of processors by Beame and Hastad[7], even if a randomized algorithm is sought (using a theorem by Adleman[6]).

However, in the HPF language, there are still some specifications should be further considered. We have a fixed number of processors in HPF, and all the data have been previously distributed among the processors in some kinds of distribution models. So we couldn’t use the general cases of algorithms to deal with the implementation of parallel prefix / suffix functions, meanwhile, we should provide an efficient algorithm to implement these functions on distributed memory systems based on message passing.

In the parallel implementation of these functions in runtime support, the array is distributed among the processors in a specific distribution model, each processor has only a part of the array elements. The prefix / suffix functions should be applied to the whole array, not only the elements in a local processor. There are different kinds of array data distribution models in HPF, such as BLOCK, CYCLIC, BLOCK-CYCLIC [1], which makes it difficult to implement these functions efficiently, especially for BLOCK-CYCLIC case. In the research and implementation of HPF language, several groups have worked on the BLOCK and CYCLIC distribution, [12] [13], but not much work on the implementation of BLOCK-CYCLIC case [14]. In this paper, we will discuss the BLOCK, CYCLIC and BLOCK-CYCLIC distribution mode of array in HPF in the following respectively, and then give an efficient parallel algorithm to implement the parallel prefix and suffix functions with high performance.

The prefix and suffix functions will all scan the array and give a kind of binary operation to each element in the array. The mechanism of prefix and suffix functions actually is the same, scanning the whole vector and associate with an operation. To simplify the discussion, we will discuss the sum_prefix operation to an array to show how to implement the prefix kind of functions in parallel.

The remainder of this paper is organized as follows. Section 2 describes the general data distribution models in HPF. In section 3, we give an general algorithm to deal with sum_prefix function of an array locally and in parallel. In section 4, we study every data distribution model and give an efficient prefix / suffix parallel algorithm to each model respectively. Section 5 will discuss the algorithm works in multi-dimension case. And finally, section 6 is the algorithm analysis and some testing experiments based on this algorithm.
2. Model

First, let’s consider the functionality of sum_prefix. Let array A has the elements \(a_1, a_2, \ldots, a_n\), then the function sum_prefix will generate the result as: \(a_1, a_1+a_2, a_1+a_2+a_3, \ldots, a_1+a_2+a_3+\ldots+a_n\). [2]

In HPF, we have the following canonical form of declaration of an array:

```hpf
REAL X(l_x : u_x)
!HPFS PROCESSORS P(N)
!HPFS TEMPLATE T(l_t : u_t)
!HPFS DISTRIBUTE T(BLOCK / CYCLIC) ONTO P
!HPFS ALIGN X(i) WITH T(a*i+b)
```

Here, we have three kinds of data distribution models, BLOCK, CYCLIC and BLOCK-CYCLIC.

2.1. BLOCK

Let array A is distributed in a BLOCK model and the elements of array A are \(a_1, a_2, \ldots, a_n\), then the pattern will be like:

```
proc 1          proc 2          proc 3          ..........          
[ a_1, a_2, ..., a_b ] [ a_{b+1}+a_{b+2}+\ldots+a_{2b} ] [ a_{2b+1}+a_{2b+2}+\ldots+a_{3b} ] ..........
```

The result of sum_prefix function will be like:

```
proc 1          proc 2          ..........          
[ a_1, a_1+a_2, ..., a_1+a_2+\ldots+a_b ] [ a_1+\ldots+a_{b+1}, \ldots, a_1+\ldots+a_{2b} ] ..........
```

2.2. CYCLIC

Let array A \((a_1, a_2, \ldots, a_n)\) is distributed in a CYCLIC model and distributed among p processors, the data distribution pattern will be like:

```
proc 1          proc 2          ......          proc p          
[ a_1, a_{p+1}, a_{2p+1}, \ldots ] [ a_2, a_{p+2}, a_{2p+2}, \ldots ] ...... [ a_p, a_{2p}, a_{3p}, \ldots ]
```

The result of sum_prefix function will be like:

```
proc 1          proc 2          ......          proc p          
[ a_1, a_1+a_2+\ldots+a_{p+1}, \ldots ] [ a_1+a_2, a_1+a_2+\ldots+a_{p+2}, \ldots ] ...... [ a_1+a_2+\ldots+a_p, a_1+a_2+\ldots+a_{2p}, \ldots ]
```

2.3. BLOCK-CYCLIC

BLOCK-CYCLIC distribution is a special data distribution mode which combines the features of BLOCK and CYCLIC distribution, and will CYCLIC distribute the array among the processors in a specific BLOCK unit. In HPF, BLOCK-CYCLIC distribution is in the form of:

```hpf
!HPFS DISTRIBUTE T(CYCLIC(n)) ONTO P
```
where \( n \) is the size of each BLOCK unit. Let's consider array \( A(a_1, a_2, ..., a_n) \) is distributed in BLOCK-CYCLIC mode, and let \( n=3 \), we will get the following data distribution pattern:

\[
\begin{align*}
\text{proc 1} & \quad \text{proc 2} & \quad \text{......} & \quad \text{proc p} \\
[ a_1, a_2, a_3, a_{3p+1}, a_{3p+2}, a_{3p+3}, ... ] & \quad [ a_4, a_5, a_6, ... ] & \quad \text{......} & \quad [ a_{3p-2}, a_{3p-1}, a_{3p}, ... ] \\
\end{align*}
\]

The result of sum_prefix function will be like:

\[
\begin{align*}
\text{proc 1} & \quad \text{proc 2} & \quad \text{......} & \quad \text{proc p} \\
[ a_1, a_1+a_2, a_1+a_2+a_3, a_1+a_2+a_3+a_{3p+1}, ... ] & \quad [ a_1+...+a_4, a_1+...+a_5, a_1+...+a_6, ... ] & \quad \text{......} & \quad [ a_1+...+a_{3p-2}, a_1+...+a_{3p-1}, ... ] \\
\end{align*}
\]

In the HPF runtime system, the array is distributed among the processors with the alignment to the template, all the operations associated with the array will be implemented to the template. In order to locate a desired element in the distributed system, we need to find its corresponding position in the global template, and then locate its position in the local processor through the array distribution model. [9]

For example: an element of array \( A \) is \( a_i \), with the alignment it is corresponding to the position in template \( T \) of \( a^i+b \), and then through the distribution mode of template among the processor, we can then calculate its position of processor and the offset in the local processor. This global_to_local and local_to_global address convert function is the basic requirement for the implementation of parallel prefix / suffix intrinsic functions. (Refer to Adlib implementation)

### 3. Parallel Sum Algorithm

#### 3.1. Local Sum_prefix Algorithm

Assume array \( A(a_1, a_2, ..., a_n) \) are stored in one processor, we can use the following algorithm to calculate sum_prefix function locally.

**Algorithm:** (Local sum_prefix)

```plaintext
Begin
1. Sum(1) = A(1)
2. For i = 2 TO n DO
   Sum(i) = Sum(i-1) + A(i)
End
```

We get the final local sum_prefix result in this new Sum array.

#### 3.2. Parallel Sum Algorithm

In HPF programs, the arrays are distributed among the processors, each processor has only a part of the elements. In order to implement the sum_prefix function along the whole array, we first consider the following parallel sum algorithm.
Let Array A \((a_1, a_2, \ldots, a_n)\) is distributed among \(n\) processors, each processor has one element of the whole array. We can do the parallel sum as the following algorithm: [10]

For \(m = 1\) to \(\lceil \log n \rceil\) DO

Begin
1. Send the data from processor \(i\) to the processor \(i + 2^m - 1\) if it exists;
2. Apply the sum operation to the element in each processor;

END

For example:

Let array A \((A_1, A_2, \ldots, A_8)\) are stored in 8 processors, through the algorithm above, we will get the result of parallel sum of the array in \(\lceil \log 8 \rceil = 3\) steps.

From this parallel sum calculation, actually we get the sum_prefix of this array with 8 elements distributed among 8 processors. In this parallel sum algorithm, the computing complexity is \(\log n\), \(n\) is the number of processors.

4. Parallel Sum_prefix Algorithm

In the implementation of HPF runtime system, the array is first aligned with the template, and then the template
is distributed among the processors in a specific distribution model. In order to simplify the discussion, we look through the effective elements in each processor and will get the following data distribution pattern:

4.1. BLOCK Distribution

```
proc 1  proc 2  proc 3
[a1, a2, ..., a_b1]  [a_{b1+1}, a_{b1+2}, ..., a_{b2}]  [a_{b2+1}, a_{b2+2}, ..., a_{b3}] .......... 
```

Each processor has a part of the elements of array A. In order to calculate the sum_prefix function, we can use the following algorithm.

Algorithm: (Sum_prefix for BLOCK distribution)
Begin:
1. Calculate the sum_prefix function locally in each processor, we will get:

```
proc 1  proc 2
[a1, a1+a2, ..., a1+a2+...+a_{b1}]  [a_{b1+1}, a_{b1+1}+a_{b1+2}, ..., a_{b1+1}+a_{b1+2}+...+a_{2b2}] .......... 
```

To simplify the expression, we rename this local sum_prefix result array as:

```
proc 1  proc 2  proc 3
[s1, s2, ..., s_{b1}]  [s_{b1+1}, s_{b1+2}, ..., s_{b2}]  [s_{b2+1}, s_{b2+2}, ..., s_{b3}] .......... 
```

2. Use parallel sum algorithm to calculate sum_prefix function.
   For m=1 to \(\lceil\log_p n\rceil\) DO
   Begin
   1. Send the last element \(s_{bi}\) in processor \(i\) to the processor \(i + 2^m - 1\) if it exists;
   2. Apply the sum operation to the whole array in each local processor;
   END
3. Then we have the sum_prefix result in this array, distributed among the processor in the same mode.

END

. Pseudocode in F90 syntax

```
! distributed arrays...
real a(p, b) ! p = num procs, b = max blocksize = (n + p - 1) / p
real result(p, b)

! per process local variables...
int locb(p) ! local blocksize <= b
real last(p), parpre(p)

! do local prefix in parallel on all processors (pseudocode!)
forall(i = 1 : p)
if(locb(i) > 0) then
   result(i, 1) = a(i, 1)
do j = 2, locb(i)
   result(i, j) = result(i, j - 1) + a(i, j)
endo
```
last(i) = result(i, locb(i))
else
  last(i) = 0
endif
endforall

! the non-local part, parallel sum.
do m = 1, \lceil \log p \rceil
  parpre = eoshift(last, -2^m - 1)
  forall(i = 1 : p)
    if(locb(i) > 0) then
      do j = 1, locb(i)
        result(i, j) = parpre(i) + result(i, j)
      enddo
      last(i) = result(i, locb(i))
    endif
  endforall
enddo

4.2. CYCLIC Distribution

Things will be quite different in CYCLIC distribution mode and will be more tricky. Let’s consider the most general case.

REAL A(l_x : u_x)
!HPFS PROCESSORS P(N)
!HPFS TEMPLATE T(l_{tx} : u_{tx})
!HPFS DISTRIBUTE T(CYCLIC) ONTO P
!HPFS ALIGN A(i) WITH T(a*i+b)

With this alignment, element \( a_i \) in the array will be located in processor \((a * i + b - l_x) \mod n\). Then the whole array will be distributed in \( \frac{lcm(a, n)}{a} = m \) processors. (lcm means least common multiple). Then suppose the first element is in processor \( p_1 \), the second element is in processor \( p_2 \), ..., we can get the array distribution pattern like: (\( p_i \) should not be the processor number i)

\[
\begin{array}{c}
\text{proc } p_1 \\
[a_1, a_{m+1}, a_{2m+1}, \ldots]
\end{array}
\begin{array}{c}
\text{proc } p_2 \\
[a_2, a_{m+2}, a_{2m+2}, \ldots]
\end{array}
\begin{array}{c}
\vdots
\end{array}
\begin{array}{c}
\text{proc } p_m \\
[a_m, a_{2m}, a_{3m}, \ldots]
\end{array}
\]

Now we can get the following parallel sum_prefix algorithm:

**Algorithm:** (Sum_prefix for CYCLIC distribution)

Begin:

1. Use the parallel sum algorithm to the array elements respectively.
   
   For \( k = 1 \) to \( \lceil \log p \rceil \) DO
   
   Begin
   
   1. Send all the elements in processor \( i \) to the processor \( p_{i+2^i-1} \) if it exists;
   
   2. Apply the sum operation to the corresponding elements in the whole array in each local processor respectively;

4
Then we will get the parallel sum of the array elements like this:

\[
\text{proc } p_1 \quad \text{proc } p_2 \quad \text{proc } p_m
\]

\[
[a_1, a_{m+1}, a_{2m+1}, ...] \quad [a_1+a_2, a_{m+1}+a_{m+2}, ...] \quad \ldots \quad [a_1+a_2+...+a_m, a_{m+1}+a_{m+2}+...+a_{2m}, ...]
\]

2. Do the local sum_prefix function in processor \( p_m \):

\[
\text{proc } p_1 \quad \text{proc } p_2 \quad \text{proc } p_m
\]

\[
[a_1, a_{m+1}, a_{2m+1}, ...] \quad [a_1+a_2, a_{m+1}+a_{m+2}, ...] \quad \ldots \quad [a_1+a_2+...+a_m, a_{m+1}+...+a_{m+2}, ...]
\]

3. Broadcast the array in the last processor \( p_m \) to the other processors, as \( b_1, b_2, ... b_s \);

4. Add the received array to the local array like: \( b_1+a_2, b_2+a_3, ... \); (except the last processor)

Then we will get the final result of sum_prefix:

\[
\text{proc } p_1 \quad \text{proc } p_2 \quad \text{proc } p_m
\]

\[
[a_1, a_1+...+a_{m+1}, a_1+...+a_{2m+1}, ...] \quad [a_1+a_2, a_1+...+a_{m+2}, ...] \quad \ldots \quad [a_1+a_2+...+a_m, a_1+...+a_{m+1}+...+a_{2m}, ...]
\]

END

**Pseudocode in F90 syntax**

```fortran
! distributed arrays ...

real a(p, b) ! p = num procs, b = (n + lcm(a, p) / a - 1) / (lcm(a, p) / a)
real result(p, b)
real received(p, b) ! containing the received array when communication

! per processor local varibles . . .

int locb(p) ! local array size <= b

! reordering the processor number according to the elements it contains,
! form a new list 1', 2', p', p' = lcm(a, p) / a
p' = reorder(a, p)

! do parallel sum
forall (i = 1 : p') ! duplicate the array a(p, b) to result(p, b)
   if (locb(i) > 0) then
      do j = 1, locb(i)
         result(i, j) = a(i, j)
      enddo
   endif
endforall

do m = 1, [\log_2 p']
forall (i = 1 : p')
   ! move the whole result array to the corresponding processor
   received = array_eoshift(result, -2^{m-1})
   if (locb(i) > 0) then
```

8
do j = 1, locb(i)
    result(i, j) = result(i, j) + received(i, j)
enddo
endif
endforall
enddo

! do local sum_prefix in the last effective processor p'
if (processor(p')) then
    do j = 2, locb(p')
        result(i, j) = result(i, j) + result(i, j-1)
    enddo
endif
forall (i = 1 : p'-1)

! Broadcast the whole array in the last processor to other processors
received = broadcast(result, p')
!
! Add the received array to the array in each processor
if (locb(i) > 0) then
    do j = 2, locb(i)
        result(i, j) = result(i, j) + received(i, j-1)
    enddo
endif
endforall

For examples:

Example 1:

Let array A a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_10, a_11, a_12 distributed in CYCLIC mode among 4 processors like:

<table>
<thead>
<tr>
<th>proc 1</th>
<th>proc 2</th>
<th>proc 3</th>
<th>proc 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_1, a_5, a_9</td>
<td>a_2, a_6, a_10</td>
<td>a_3, a_7, a_11</td>
<td>a_4, a_8, a_12</td>
</tr>
</tbody>
</table>

Step 1: After the parallel sum calculation, we will get:

<table>
<thead>
<tr>
<th>proc 1</th>
<th>proc 2</th>
<th>proc 3</th>
<th>proc 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_1, a_5, a_9</td>
<td>a_1+a_2, a_5+a_6, a_9+a_{10}</td>
<td>a_1+a_2+a_3, a_5+a_6+a_7, a_9+a_{10}+a_{11}</td>
<td>a_{1-4}, a_5, a_9, a_{12}</td>
</tr>
</tbody>
</table>

Step 2: After the local sum_prefix function in processor 4, we can get:

<table>
<thead>
<tr>
<th>proc 1</th>
<th>proc 2</th>
<th>proc 3</th>
<th>proc 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_1, a_5, a_9</td>
<td>a_1-2, a_5-6, a_9-10</td>
<td>a_1-3, a_5-7, a_9-11</td>
<td>a_{1-4}, a_1-8, a_{1-12}</td>
</tr>
</tbody>
</table>

Step 3: Broadcast the array in processor 4 to the other three processors;

Step 4: Add the received array to the local array separately, we can then get the final parallel sum_prefix result:

<table>
<thead>
<tr>
<th>proc 1</th>
<th>proc 2</th>
<th>proc 3</th>
<th>proc 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_{1-5}, a_{1-9}</td>
<td>a_{1-2}, a_{1-6}, a_{1-10}</td>
<td>a_{1-3}, a_{1-7}, a_{1-11}</td>
<td>a_{1-4}, a_{1-8}, a_{1-12}</td>
</tr>
</tbody>
</table>
Now, let’s consider a more complex example.

Example 2:

```
REAL A(1 : 12)
!HPFS PROCESSORS P(5)
!HPFS TEMPLATE T(1 : 100)
!HPFS DISTRIBUTE T(CYCLIC) ONTO P
!HPFS ALIGN A(i) WITH T(3*i-1)
```

In this example, the distribution pattern of array A will be like:

```
proc 1     proc 2     proc 3     proc 4     proc 5
.. a4 .. a9 ..    a1 .. a6 .. a11 ..    a3 .. a8 ..    a5 .. a10 ..    a2 .. a7 .. a12 ..
```

**Step 1:** According to the algorithm, we first select the effective elements in each processor to form a new array and reorder the processor number according to the elements it contains. Then we will get the following pattern:

```
proc 1'   proc 2'   proc 3'   proc 4'   proc 5'
[ a1, a6, a11 ]  [ a2, a7, a12 ]  [ a3, a8 ]  [ a4, a9 ]  [ a5, a10 ]
```

**Step 2:** Apply the parallel sum algorithm to this new pattern, we will get:

```
proc 1'   proc 2'   proc 3'   proc 4'   proc 5'
[ a1, a6, a11 ]  [ a1,2, a6,7, a11,12 ]  [ a1,3, a6,8 ]  [ a1,4, a6,9 ]  [ a1,5, a6,10 ]
```

**Step 3:** Apply the local sum_prefix function to the array in the last processor 5’;

```
proc 1'   proc 2'   proc 3'   proc 4'   proc 5'
[ a1, a6, a11 ]  [ a1,2, a6,7, a11,12 ]  [ a1,3, a6,8 ]  [ a1,4, a6,9 ]  [ a1,5, a1,10 ]
```

**Step 4:** Broadcast the array in processor 5’ to the other four processors;

**Step 5:** Add the received array to the local array seperately, we can then get the final parallel sum_prefix result:

```
proc 1'   proc 2'   proc 3'   proc 4'   proc 5'
[ a1, a1,6, a1,11 ]  [ a1,2, a1,7, a1,12 ]  [ a1,3, a1,8 ]  [ a1,4, a1,9 ]  [ a1,5, a1,10 ]
```

### 4.3. BLOCK-CYCLIC Distribution

BLOCK-CYCLIC distribution has the features of both BLOCK and CYCLIC. It first separates the whole array...
in fixed sized BLOCK units, then distributes the units among the processors in CYCLIC mode. And because of
this property, its parallel sum_prefix function becomes more complex. Let’s consider the most general case.

REAL A(lx : ux)
!HPF$ PROCESSORS P(N)
!HPF$ TEMPLATE T(ltx : utx)
!HPF$ DISTRIBUTE T(CYCLIC(k)) ONTO P
!HPF$ ALIGN A(i) WITH T(a*i+b)

In this array alignment, element \(a_i\) in array \(A\) will be located in processor \(((a*i+b-lx)/k) \mod n\). And the corre-
sponding position of the local template will be: \(((a*i+b-lx)/k)/n)*(k-1)+((a*i+b-lx) \mod k)\).

In order to deal with sum_prefix function in this kind of distribution, we first separate the local template into
BLOCK units, each unit has \(k\) elements of the global template. The value of the last template element in each
BLOCK unit which has no array element aligned with will be set to 0. This is for the purpose of storing the tem-
porary value when applying the parallel sum algorithm.

First, we deal with the BLOCK distributed mode part elements using BLOCK mode algorithm, and then use the
CYCLIC mode algorithm to treat the CYCLIC distribution data. We will get the algorithm like:

**Algorithm:** (Sum_prefix for BLOCK-CYCLIC distribution)

Begin:
1. Apply the local sum_prefix algorithm to each BLOCK unit in each processor;
2. Use parallel sum algorithm to calculate sum_prefix function of each corresponding BLOCK unit.
   For \(m=1\) to \(\lceil \log n \rceil\) DO
   Begin
   1. Send the last elements of BLOCK unit \(j\) in processor \(i\) to the BLOCK unit \(j\) of processor \(i + 2^m - 1\)
      if it exists;
   2. Add the data received to all the elements of each BLOCK unit which has effective array element
      aligned with in each processor (including the last element);
   END
   (Finish the first part of dealing with the BLOCK distribution mode.)
3. Clear the last element of each BLOCK unit which has no effective array element aligned with in all proces-
sors, except the last processor;
4. Do the local sum_prefix function in the last processor using the last elements of each BLOCK unit;
5. Use the last element of each BLOCK unit in the last processor to form a new array, as \(b_1, b_2, \ldots, b_s\), and
   then broadcast the array to the other processors;
6. Add the received array to the local BLOCK unit like: \(b_1+B_2, b_2+B_3, \ldots\) (except the last processor), \(B_i\) is the
   \(i\)th BLOCK unit;
7. Clear the last element of each BLOCK unit where there’s no effective array element aligned with in the last
   processor;
END

**Pseudocode in F90 syntax**

```f90
! distributed arrays ...
```
real a(p, b, c) ! p = num procs, c = BLOCK unit SIZE, b = (n + c*p - 1) / (c*p)
real result(p, b, c) ! initialized to 0

! per processor local variables ...

int num_block = b ! max BLOCK unit number
int unit_size = c ! size of each BLOCK unit
real last(p, b), received(p, b)

! do local sum_prefix in parallel in each BLOCK unit on all processors
forall (i = 1 : p)
do j = 1, num_block
   result(i, j, 1) = a(i, j, 1)
do k = 2, unit_size
   result(i, j, k) = result(i, j, k-1) + a(i, j, k)
endo
last(i, j) = result(i, j, unit_size)
endo
endforall

! do parallel sum.
do m = 1, \[\log_2 p\]
   ! move the last array to the corresponding processor
   received = array_eoshift(last, -2^{m-1})
   ! add the received array to each corresponding BLOCK unit
forall (i = 1 : p)
do j = 1, num_block
   do k = 1, unit_size
      result(i, j, k) = result(i, j, k) + received(i, j)
   enddo
last(i, j) = result(i, j, unit_size)
endo
endforall
endo

! clear the elements which has no array element aligned with in each processor,
! except the last processor
forall (i = 1 : p-1)
do j = 1, num_block
   do k = 1, unit_size
      ! effective(x) is to check if a(i, j, k) is an effective array element
      ! if (effective(a(i, j, k))) then
      result(i, j, k) = 0
      endif
   enddo
endo
endforall

! do local sum_prefix in the last processor
if (processor(p)) then
   do j = 2, num_block
      do k = 1, unit_size
         result(p, j, k) = result(p, j, k) + last(p, j-1)
      enddo
last(p, j) = result(p, j, unit_size)
   enddo
endif
endif
forall (i = 1 : p-1)
! Broadcast the whole last array in the processor p to other processors
received = broadcast(result, p)
! Add the received array to the array in each processor
do j = 2, num_block
do k = 1, unit_size
  if (effective(a, j, k)) then
    result(i, j, k) = result(i, j, k) + received(i, j-1)
  endif
enddo
enddo
endforall
! clear the elements which has no array element aligned with in processor p.
if (processor(p)) then
do j = 1, num_block
do k = 1, unit_size
  ! check if a(i, j, k) is an effective array element
  if (!effective(a(p, j, k))) then
    result(p, j, k) = 0
  endif
enddo
enddo
endif

For examples:

Example 1:

REAL A(1 : 20)
!HPF$ PROCESSORS P(4)
!HPF$ TEMPLATE T(1 : 100)
!HPF$ DISTRIBUTE T(CYCLIC(3)) ONTO P
!HPF$ ALIGN A(i) WITH T(i)

In this example, the array distribution pattern is like:

proc 1  proc 2  proc 3  proc 4
[ (a_1, a_2, a_3), (a_{13}, a_{14}, a_{15}) ]  [ (a_4, a_5, a_6), (a_{16}, a_{17}, a_{18}) ]  [ (a_7, a_8, a_9), (a_{19}, a_{20}) ]  [ (a_{10}, a_{11}, a_{12}) ]

Step 1: Apply local sum_prefix to each BLOCK unit, then we will get:

proc 1  proc 2
[ (a_1 + a_2, a_1 + a_2 + a_3), (a_{13} + a_{14} + a_{15}) ]  [ (a_4 + a_5, a_4 + a_5 + a_6), (a_{16} + a_{17} + a_{18}) ]
Step 2: Apply the parallel sum_prefix to each corresponding BLOCK unit among the processors (sending the last element in each BLOCK unit):

Step 3: Use the last element in the BLOCK unit of the last processor to form a new array, we get \( a_{1-12} \), and then broadcast it to the other three processors;

Step 4: Add \( a_{1-12} \) to the second BLOCK unit of each processor if existed;

Then we get the final sum_prefix result:

Examples 2:

```plaintext
REAL A(1 : 15)
!HPF$ PROCESSORS P(5)
!HPF$ TEMPLATE T(1 : 100)
!HPF$ DISTRIBUTE T(CYCLIC(2)) ONTO P
!HPF$ ALIGN A(i) WITH T(3*i-1)
```

In this example, the array A is aligned with global template T and distributed among the processors like:

Step 1: Apply local sum_prefix to each BLOCK unit, then we will get:
Step 2: Apply the parallel sum_prefix to each corresponding BLOCK unit among the processors (sending the last element in each BLOCK unit, some position of last element in the BLOCK unit with no array element aligned with maybe assigned with a temporary value for the purpose of implementation of the parallel sum algorithm);

Step 3: Clear the temporary value of the last element which has no array element aligned with in each BLOCK unit in each processor, except the last processor;

Step 4: Apply the local sum_prefix function to the array in the last processor (proc 5), using the last element of each BLOCK unit when adding;

Step 5: Use the last element in the BLOCK unit of the last processor to form a new array, we get: \( a_{1-3}, a_{4-7}, a_{8-10}, a_{11-13}, a_{14-15} \), and then broadcast the new array to the other processors;

Step 6: Add the received array to the local BLOCK unit separately in each processor, except the last processor;
Step 7: Clear the temporary value of the last element which has no array element aligned with in each BLOCK unit in the last processor;

5. Multi-Dimensional Cases

Now we survey the multi-dimensional parallel sum_prefix function. There are two kinds of basic parallel sum_prefix functions in HPF. To simplify the discussion, we first discuss a two dimension example.

Let A is the array
\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix}
\]

SUM_PREFIX(A) is the array
\[
\begin{bmatrix}
1 & 14 & 30 \\
5 & 19 & 36 \\
12 & 27 & 45
\end{bmatrix}
\]

SUM_PREFIX(A, DIM=1) is the array
\[
\begin{bmatrix}
1 & 2 & 3 \\
5 & 7 & 9 \\
12 & 15 & 18
\end{bmatrix}
\]

SUM_PREFIX(A, DIM=2) is the array
\[
\begin{bmatrix}
1 & 3 & 6 \\
4 & 9 & 15 \\
7 & 15 & 24
\end{bmatrix}
\]

The first kind is to apply the sum_prefix function along the whole array, and the last two is to calculate the sum_prefix along one specific dimension of the array.
To implement the multi-dimension array sum_prefix function, we first gather the corresponding elements in the array to form one or more new one dimension arrays, then use the algorithm we just discussed before to each one dimension array to get the final parallel sum_prefix result.

In the above example, we should form an one dimension array as B (1, 2, 3, 4, 5, 6, 7, 8, 9), then according the alignment of array A to the template T, and the distribution of template T, use the parallel sum_prefix algorithm to this new array B. After we get the result, we then reform the one dimension result into a two dimension array according to the mapping between A and B.

And to deal with the sum_prefix along a specific dimension of A (the later two cases), we will form three one-dimension array like B1 (1, 4, 7), B2 (2, 5, 8), B3 (3, 6, 9) for DIM=1, and C1 (1, 2, 3), C2 (4, 5, 6), C3(7, 8, 9) for DIM=2, respectively. And then use the parallel sum_prefix algorithm above to calculate their sum_prefix function, then reform the result two-dimension array from the mapping between A and B1, B2, B3, and/or A and C1, C2, C3.

From this kind of linearized approach, we can implement the parallel sum_prefix function of multi-dimensional array in the Runtime support for HPF.

6. Algorithm analysis and Experiments

Let’s consider how much operations we need to calculate the parallel sum_prefix function. Suppose the array size is N, and we have P processors. Then we need N add operations to implement the prefix function sequentially. Now consider the parallel case according to the distribution models.

. BLOCK

Each processor will have N/P elements, and suppose Comm1 is the communication operation of one element, then the total operations will be:

\[ \frac{N}{P} + \log P \cdot \text{Comm}_1 + \frac{N}{P} = \frac{2N}{P} + \log P \cdot \text{Comm}_1 \]

. CYCLIC

\[
\log P \cdot \text{Comm}_1 \left( \frac{N}{P} \right) + \frac{N}{P} + \frac{N}{P} + \text{Broadcast} \left( \frac{N}{P} \right) + \frac{N}{P} \\
= 3\frac{N}{P} + \log P \cdot \text{Comm}_1 \left( \frac{N}{P} \right) + \text{Broadcast} \left( \frac{N}{P} \right)
\]

. BLK-CYC

Suppose the size of each BLOCK is B, then the total operations will be:

\[ B + \log P \cdot \text{Comm}_1 \left( \frac{N}{(B \cdot P)} \right) + \frac{N}{(B \cdot P)} + 2\frac{N}{P} + \text{Broadcast} \left( \frac{N}{(B \cdot P)} \right) \]

We got the testing result of this algorithm on DEC Alpha farms parallel machines like:
Table 1: BLOCK Mode

<table>
<thead>
<tr>
<th>Processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array size = 1M</td>
<td>0.775s</td>
<td>1.032s</td>
<td>0.530s</td>
<td>0.363s</td>
<td>0.283s</td>
</tr>
</tbody>
</table>

Table 2: CYCLIC Mode

<table>
<thead>
<tr>
<th>Processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array size = 100K</td>
<td>0.109s</td>
<td>0.148s</td>
<td>0.128s</td>
<td>0.207s</td>
<td>0.162s</td>
</tr>
</tbody>
</table>

Table 3: BLK_CYC mode

<table>
<thead>
<tr>
<th>Processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array size = 100K</td>
<td>0.228s</td>
<td>0.191s</td>
<td>0.159s</td>
<td>0.130s</td>
<td>0.119s</td>
</tr>
</tbody>
</table>

Reference:


[5]. S. Lakshmivarahan, C. M. Yang, and S. K Dhall. Optimal Parallel Prefix Circuits with \( (\text{size} + \text{depth}) = 2n-n \) and \( \lfloor \log_{n} \rfloor \leq \text{depth} \leq \lfloor 2\log_{n} \rfloor - 3 \). In Proceedings International Conference on Parallel Processing, pp 58-65, August 1987.

[6]. Leonard M. Adleman, Two theorems on random polynomial time, In proceedings of the 19th IEEE Annual Symp. on Foundation of Computer Science, pp 75-83, 1987

[7]. Paul Beame and Johan Hastad, Optimal bounds for decision problems on the CRCW PRAM, In Proc. of


