Asynchronous Progressive Irregular Prefix Operation in HPF2

Frédéric Brégier, Marie-Christine Counilh, Jean Roman
LaBRI, ENSERB et Université Bordeaux I, 33405 Talence Cedex, France
Published in an IEEE conference act : PDP'2000, IEEE copyrighted

Abstract

In this paper, we study one kind of irregular computation on distributed arrays, the irregular prefix operation, *that is currently not well taken into account by the standard data-parallel language HPF2. We show a parallel implementation that efficiently takes advantage of the independent computations arising in this irregular operation. Our approach is based on the use of a directive which characterizes an irregular prefix operation and on inspector/executor support, implemented in the* CoLuMBO *library, which optimizes the execution by using an asynchronous communication scheme and then communication/computation overlap. We validate our contribution with results achieved on IBM SP2 for basic experiments and for a sparse Cholesky factorization algorithm applied to real size problems.*

KEY WORDS: *HPF2, irregular application, prefix operation, run-time support, inspection/execution mechanism, loop-carried dependencies*

1. Introduction

High Performance Fortran (HPF2 [10]), the standard language for writing data parallel programs, is quite efficient for regular applications. Nevertheless, efficiency is still a great challenge when irregular applications are considered.

In this paper, we study one kind of irregular computation on distributed arrays that we call irregular prefix operations. This kind of computation occurs in important irregular algorithms such as sparse Cholesky factorization.
Our goal is to propose a parallel implementation of this irregular operation that efficiently takes advantage of the independent computations arising in it. This implementation is based first on the use of a directive which specifies that a loop performs an irregular prefix operation; second, on an inspector/executor support, implemented in the C*o*L*u*MBO library, which optimizes the execution phase by using an asynchronous communication scheme and then communication/computation overlap.

This paper is organized as follows. In section 2, we define an irregular prefix operation on a vector. We present different ways of writing it in HPF2 and show the limits of these versions. Then, we present our `PREFIX` clause and directive for irregular prefix operation and we describe our implementation based on the inspection/execution approach. Finally, we present some related works. Section 3 describes our experimental work on IBM SP2. We present first some basic experiments in order to study and analyze the contribution of our approach. Then, we present an experimental study for sparse Cholesky factorization applied on real size problems, which confirms its interest. Finally, section 4 concludes and gives some perspectives to our work.

2. Progressive Irregular Prefix Operation

2.1. Definition

A prefix operation on a vector is an operation where each element of the output vector is a function of the elements of the input vector that precede it. For instance, the prefix sum of the input vector C of size N is the output vector X where:

$$\forall i \in [1, N] \;\; X_i = \sum_{1 \leq k \leq i} C_k \; .$$

Prefix operations are very useful in data parallel programming and they have been included in the HPF library. So, efficient parallel implementations of these operations are possible.

An *irregular prefix operation* is such that each element of the output vector is a function of an arbitrary subset of the elements of the input vector that precede it. For example:

$$\forall i \in [1, N] \;\; X_i = \sum_{k \in B_i} C_k \text{ where } B_i \subseteq [1, i] \; .$$

We define a *progressive irregular prefix operation* on one vector as an irregular prefix operation, where the input and

output vectors are the same. More precisely, each output element $X(i)$ depends on the *output* values of the elements $X(k)$, $k \in B_i$ and $k < i$, and on the input value $X(i)$. So these values are computed by step (in worst case, step by step), so the *progressive* attribute. Moreover, functions can be applied to each element during the prefix operation (g) and to each element of the result (f). For instance:

$$\forall i \in [1, N] \;\; X_i = f(\sum_{k \in B_i} g(X_k)) \text{ where } B_i \subseteq [1, i] \; .$$

Progressive irregular prefix operations on distributed vectors (more generally arrays) are useful in irregular applications, for example in sparse matrix applications such as sparse Cholesky factorization. Our goal is to propose a parallel implementation that efficiently takes advantage of independent computations (if $j \notin B_i$ and $i \notin B_j$, then X_i and X_j can be computed in parallel).

2.2. Progressive Irregular Prefix Operation in HPF2

In this section, we present different ways of writing an irregular prefix operation in HPF2 and we show the limits and disadvantages of each of these versions.

Program 1 shows two Fortran codes with two nested DO loops. In Program 1(a), the B_i $(i = 1, N)$ sets are replaced by a 2-dimensional array `B` where `B(i,:)` contains the `NB(i)` elements of the $B_i - \{i\}$ set. Program 1(b) uses a symmetric approach by using the 2-dimensional array `Bs` such that: `K` $\in$ `Bs(I,:)` $\Leftrightarrow$ `I` $\in$ `B(K,:)` (so, elements in `Bs(I,:)` belong to `]I,N]`). In these programs, the `I` and `J` loops are not `INDEPENDENT` loops because each element of `X` is computed from some of the elements of `X` that precede it. These loops are no more `INDEPENDENT` loops with reduction statement since a reduction variable cannot be used in a reduction statement (more precisely, the reference `X(K)` is forbidden in the statement `X(I) = X(I) + X(K)`; cf. [10, pp. 71-76]). So, without any information about the properties of the loops, a HPF compiler will generate an inefficient serial SPMD code based on the *owner computes rule*.

In Program 2, we use a temporary variable (`TMP`) so that the inner loop `J` is an `INDEPENDENT` loop with reduction. This loop can be simply and efficiently implemented: each processor uses a private accumulator variable associated with the reduction variable and performs a subset of the `J` loop iterations. When it encounters a reduction statement, it updates its own accumulator variable. After the loop, the final value of the reduction variable is computed by combining the private accumulator variables using the reduction operator. In an MPI based implementation, the `MPI_Reduce` function could be used for this *combining operator*. But due to the indirect access (`X(B(I,J))`), the compiler will consider that all the processors take part in the collective communication at each iteration `I` even though it

```
DO I = 1, N                    (a)
   DO J = 1, NB(I)
      X(I) = X(I) + g(X(B(I,J)))
   END DO
   X(I) = f(X(I))
END DO

DO I = 1, N                    (b)
   X(I) = f(X(I))
   DO J = 1, NBs(I)
      X(Bs(I,J)) = X(Bs(I,J)) + g(X(I))
   END DO
END DO
```

Program 1. Prefix sum (a) and prefix sum in symmetric form (b)

is not always necessary. So processors are synchronized at every iteration.

```
DO I = 1, N                    (a)
   TMP = 0.0
!HPF$ INDEPENDENT, REDUCTION(TMP)
   DO J = 1, NB(I)
      TMP = TMP + g(X(B(I,J)))
   END DO
   X(I) = X(I) + TMP
   X(I) = f(X(I))
END DO

DO I = 1, N                    (b)
   TMP = 0.0
   TMP2 = 0.0
   DO J = 1, NB(I)
      if (X(B(I,J)) is local) then
         TMP2 = TMP2 + g(X(B(I,J)))
      end if
   END DO
   REDUCTION(TMP2, SUM)
   TMP = TMP + TMP2
   if (X(I) is local) then
      X(I) = X(I) + TMP
      X(I) = f(X(I))
   end if
END DO
```

Program 2. Prefix sum with reduction (a) and its SPMD code (b)

Finally, the array prefix functions (`XXX_PREFIX`) and scatter functions (`XXX_SCATTER`) of the HPF library are not well suited to express irregular prefix operations and successive calls to these functions with appropriate mask vectors should be required. For example, the mask array that can be used in a prefix function to specify the elements of the vector that contribute to the result is not enough: here, each element of the result requires a specific mask (that represents the subset B_i).

None of the versions presented here can lead to a parallel implementation that allows parallel computations on some elements of the result. In the following section, we introduce a directive and a clause that may help a compiler to do so.

2.3. The PREFIX Clause and Directive

The `PREFIX(`*prefix-variable*`)` directive can precede a `DO` loop. It asserts the compiler that iterations in the following `DO` loop compute *prefix-variable* as the result of an (*irregular*) prefix operation.

The `PREFIX(`*prefix-variable*`)` clause is used with an `INDEPENDENT` directive and it asserts that the named variable is update in the `INDEPENDENT` loop by a series of operations that are associative and commutative. This clause is always relative to a declared surrounding `PREFIX DO` loop. The syntax of *prefix-variable* and *prefix-statement* is the same as *reduction-variable* and *reduction-statement* defined in HPF2. The difference between the `REDUCTION` and `PREFIX` clauses is that a reference to a `PREFIX` variable can occur in the operand part of a `PREFIX` statement. But, in this case, the `PREFIX` clause of the `INDEPENDENT` loop asserts the compiler that only final values (and not intermediate values) of the `PREFIX` variable are used within this loop. We consider Program 1 Version b. In this program, we know that `X(I)` gets its final value at iteration `I` (in the statement `X(I) = f(X(I))`), and that `X(I)` is never read before iteration `I` nor modified after this iteration. So we can use the `PREFIX` clause and directive to obtain Program 3.

```
!HPF$ PREFIX(X)
  DO I = 1, N
     X(I) = f(X(I))
!HPF$ INDEPENDENT, PREFIX(X)
     DO J = 1, NBs(I)
        X(Bs(I,J)) = X(Bs(I,J)) + g(X(I))
     END DO
  END DO
```

Program 3. HPF2/PREFIX code for an irregular prefix operation

In the following section, we show how a compiler can use the directive and clause introduced here.

2.4. Code Generation for a Prefix Do Loop

To implement `INDEPENDENT` loop with prefix statement, we use the same mechanism as for reductions (cf. section 2.3) by using a private accumulator variable that has the same shape than the `PREFIX` variable. For Program 3, on each processor, the private accumulator (cf. Program 4) is an array (`TMP(1:N)`). Unlike a reduction implementation, the combining operation is performed for only one element at a time of the prefix array variable. More precisely, the combining operation which computes the final value of `X(I)` from the local variables `TMP(I)` (`Reduction (TMP(I), SUM)`) must be performed within the external `PREFIX` loop; moreover, it must be executed after the last write access to each private variable `TMP(I)` and before the first read access to `X(I)`.

In some simple cases, the compiler can determine the position of the combining operation. In Program 3, it may be performed at the beginning of iteration `I` since `X(I)` is read for the first time at the beginning of this iteration and `TMP(I)` is written only before iteration `I` (according to the `PREFIX` clause). But, in Program 4, this combining operation is still performed in a synchronous way. So, we propose to introduce asynchronism in communication in order to overlap communication by computation, and more precisely, to separate the send calls and the receive calls associated with the combining operation for `X(I)` so that:

1. the send call is performed independently and as soon as possible on each processor (i.e. when the corresponding private variable `TMP(I)` has its own final value),

2. the receive calls are performed as late as possible by the processor that owns `X(I)` (i.e. before its first read operation of `X(I)` outside a prefix statement).

```
TMP(1:N) = 0.0
DO I = 1, N
   REDUCTION(TMP(I), SUM)
   if (X(I) is local) then
      X(I) = X(I) + TMP(I)
      X(I) = f(X(I))
      DO J = 1, NBs(I)
         TMP(Bs(I,J)) = TMP(Bs(I,J)) + g(X(I))
      END DO
   end if
END DO
```

Program 4. SPMD pseudo-code for Program 3

To enable such an execution, an inspection step is required to determine when local computations on `TMP(I)` are over so that it can be sent, and which processor subset effectively contributes to the final value of `X(I)` (the owner of `X(I)` must receive only from these processors).

The inspection step (cf. Program 5(a)) consists in three parts. The first one only registers an array as a *prefix variable* (`Insp_Prefix_Data`). The second part is a local one: each processor scans its local accesses to the *prefix variable*. The calls to `Insp_Prefix_Statement(X(Bs(I,J)))` count the read/write accesses to `X(Bs(I,J))` within the *prefix*

statement; the `Insp_Prefix_Access(X(I))` call determines which processor needs the final value of `X(I)`. Finally, the third part (`Insp_Prefix_Done`) is in two steps: first, all processors gather their local information by using global communications; then each processor locally keeps only the relevant information.

```
Insp_Prefix_Data(X(1:N))                    (a)
DO I = 1, N
  if (X(I) is local) then
    Insp_Prefix_Access(X(I))
    DO J = 1, NBs(I)
      Insp_Prefix_Statement(X(Bs(I,J)))
    END DO
  end if
END DO
Insp_Prefix_Done()

DO I = 1, N                                 (b)
  if (X(I) is local) then
    Prefix_Access(X(I))
          !receive: combining operation
    X(I) = f(X(I))
    DO J = 1, NBs(I)
      TMP(Bs(I,J)) = TMP(Bs(I,J)) + g(X(I))
      Prefix_Statement(X(Bs(I,J)))
          !send when ready
    END DO
  end if
END DO
```

Program 5. Inspector(a) / Executor(b) SPMD pseudo-code of Program 3

The execution step (cf. Program 5(b)) uses two routines: `Prefix_Statement` and `Prefix_Access`. Each call to `Prefix_Statement(TMP(Bs(I,J)))` decreases the counter resulting from the `Insp_Prefix_Statement(X(Bs(I,J)))` calls in the inspector step. When this counter reaches zero, the processor sends the value of its private accumulator variable `TMP(Bs(I,J))`. The `Prefix_Access(X(I))` call performs the receive operations and combines the received values to yield the final value of `X(I)`. Only one combining operation can appear for each prefix variable element (according to the `PREFIX` clause). So only the first `Prefix_Access` call for each element is required. Since all following accesses are on the same *prefix variable* element `X(I)` (excepting the access to the *prefix variable* for the *prefix statement*), the compiler can optimize the executor code by removing the subsequent `Prefix_Access` calls (which could have take place before the *prefix statement*). Anyway, all subsequent calls will be ignored by the run-time support.

So, the inspection/execution steps of Program 5 enable an "asynchronous execution of the prefix operation" where communication can be overlapped by computation. Note that Program 1(a) could also be written by using the `PREFIX` clause and directive. But, in this program, the last write access to the accumulator variable `TMP(I)` takes place just before the first read access to `X(I)`. So, in the corresponding SPMD code, the send and receive operations would take place side by side after the `ENDDO` of the `J` loop and before the statement `X(I) = f(X(I))`. Consequently, this version presents no overlap capabilities and looks like Program 2.

Note that if the global loop is declared as `PREFIX` but the `INDEPENDENT, PREFIX(X)` directive on the internal loop is omitted, the same inspection/execution scheme can be applied with a slight difference: the basic communication scheme is a broadcast (not a reduction) since the compiler will apply the *owner computes rule* on the internal loop (for Program 3 without the internal directive, `X(I)` will be broadcast). For simplicity, we omit this case in this paper.

We have implemented this inspection/execution scheme in our C*o*L*u*MBO library. We have incorporated three implementation optimizations that are not shown, for simplicity, in the SPMD codes. The first optimization limits the size of the private accumulator array to its relevant part saving memory space. The second optimization consists in the scan of sections instead of single elements of the prefix variable array so as to save memory space and time. The third optimization consists in the reception of pending communications before the corresponding `Prefix_Access` call in order to avoid the saturation of the MPI communication buffers.

2.5. Related Work

To our knowledge, progressive irregular prefix operations have not been studied in the context of HPF or HPF-like languages as FortranD [16], Vienna Fortran [14] or HPF+ [3].

Inspector/executor paradigm has been widely used but for solving iterative irregular problems in which communication and computation phases alternate; indeed, in those kinds of applications, the cost of optimizations performed at the inspector stage can be amortized over many computation iterations at the executor stage. Major works include PARTI [16] and CHAOS [11] libraries used in Vienna Fortran and Fortran90D [15] compilers, and PILAR library [12] used in PARADIGM compiler [2]. These libraries are based on a `gather/scatter` approach and use the same optimized communication scheme on every (or at least many) iteration. So they do not address such asynchronous prefix operation since each iteration of the `PREFIX` loop has its own communication scheme.

The PILAR [13] library uses sections as minimal inspected elements, as our C*o*L*u*MBO library does. This is

necessary in order to produce an efficient inspector in most cases, since the minimal element considering in Fortran programs is often, at least, a section of array and not a single element.

3. Experimental Validations

3.1. Basic Experiments

This section studies the interest of our approach with some experimental validations achieved from the three SPMD codes given at Program 2, Program 4 and Program 5. Note that for Program 5, all measures will include inspector times. In our experimentation, the value of N is 1200 and `X` is a 2-dimensional array (300×1200) and has a `(*, CYCLIC)` distribution. The costs of functions `f` and `g` vary from 300 Flops to 400 KFlop. We also use various coefficients c of irregularity characterized by the ratio (in percent) of the number of elements in each subset B_i with the corresponding maximal possible number: so $c = 100\%$ means i elements in Bi ($B_i = [1, i]$ and `B(I,J) = J, 1≤J<I`); in general, c means $i \times c$ elements in B_i. So the lower the coefficient, the lower the irregularity or the number of elements in B_i.

We first compare Program 2 and Program 5 with a fixed cost for functions `f` and `g` (cost = 10 KFlop). We measure the time ratio of Program 5 compared to Program 2 both executed with various combinations of processor number `p` and coefficient `c`. Figure 1 gives the results achieved for `p = 4, 8, 16` and `c` from 100 to 1%. For a fixed number of processors and a fixed function (`f` and `g`) cost, we define the *prefix balance coefficient* as the coefficient of irregularity such that Program 2 and Program 5 have the same execution time (time ratio $= 1.0$) (for example, this prefix balance coefficient is 4.6% for 4 processors and function cost = 10KFlop).

First of all, we can see that the lower the coefficient, the better the results of asynchronous prefix version (time ratio from 2.0 to 0.5). For Program 4, the time ratios compared to Program 2 (not shown) are always bad (from 6.0 with $c = 100\%$ to 2.0 with $c = 1\%$). This result shows the great interest of the asynchronous communication approach for the combining operation.

Figure 2(a) gives the prefix balance coefficient for various combinations of the processor number and function cost. For a fixed computation cost, one sees that increasing the number of processors increases the corresponding prefix balance coefficient. The reason is that the number of implied communications increases when more processors are involved, so there is more capability of overlapping communications by computations.

If we fixed the number of processors, so increasing the elementary cost, the number of communications does not change but computations can better overlap communications since there are more expensive. To obtain again the prefix balance coefficient, we can therefore have more communications, so, with a fixed number of processors, the higher the elementary cost, the higher the prefix balance coefficient. We also observe that the prefix balance coefficient becomes stable (between 7 and 10%) after a certain elementary cost (40 KFlop). We suppose that this stability is probably due to some overlap limitations or to some basic example specificities.

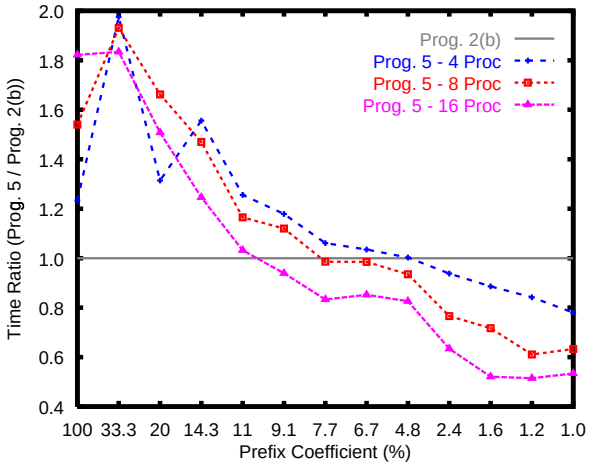

Figure 1. Time ratios vs. Prog. 2 with various coefficients of irregularity and processors ((`f`, `g`) fixed cost (10KFlop))

Another contradiction appears with the execution on 16 processors. It is in fact due to the lack of computation on so many processors, which introduces a scalability problem. Figure 2(b) presents the relative efficiencies of Program 5 with 8 and 16 processors compared to 4 processors with a coefficient of irregularity equals to the balanced prefix coefficient on 4 processors. With 0.3 and 1 KFlop and such low coefficients (2.6 and 1.8% respectively), there are low efficiencies (around 30%) on 16 processors. The computations have a very low cost, so only communications are taking into account in the relative efficiencies. Therefore, we measure in fact the relative efficiencies of asynchronous communications compared to synchronous ones. In order to obtain again the balanced prefix coefficient when there is a scalability problem, we must increase the amount of computations by increasing the coefficient, so the results on 16 processors.

Finally, we have measured the inspection cost with respect to a global execution. In our experiments, the inspection time represents between 1 to 8% of the global execution time. These good performances show the interest of the implementation optimizations performed in the inspector (cf. 2.4).

Shortly, these basic experiments show first that the lower the coefficient of irregularity, the better the performance of

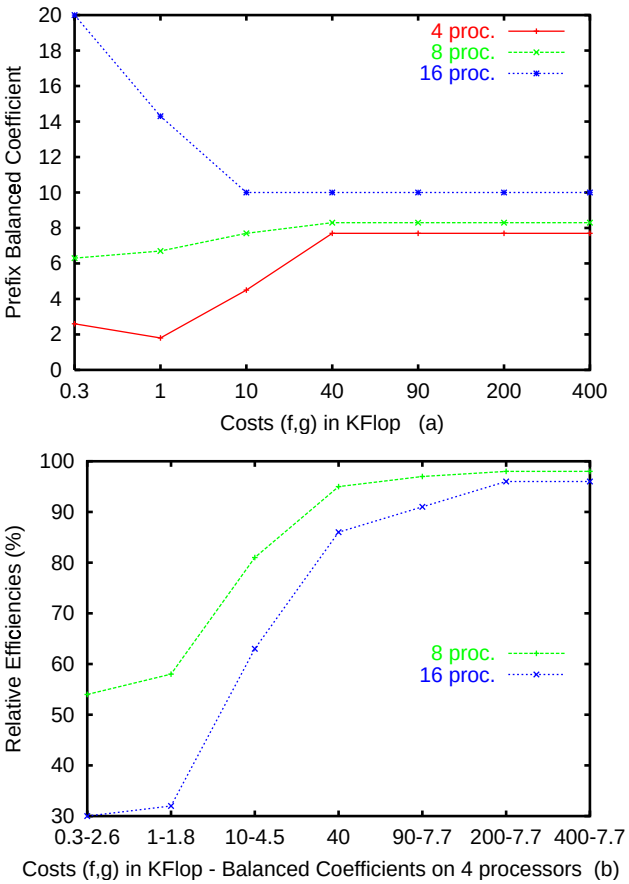

Figure 2. Prefix balance coefficients on various processors with various (f, g) costs (a) and Relative efficiencies compared to 4 processor times with fixed (f, g) costs (b)

asynchronous prefix version, and second that efficiency is achieved even with only one executor step (unlike a standard inspector/executor approach which requires many executor steps to repay the inspector cost).

3.2. Cholesky Example

We describe here the experimental results we have obtained for an important computation arising in many scientific and engineering applications: the sparse Cholesky block factorization [8, 7, 1]. In this application, each column block is updated by some column blocks to its left on the sparse matrix; the subset $B_i \subseteq [1, i]$ of column blocks depends here on the sparsity structure of the matrix. More precisely, the structure of the application is closed to the extended HPF program presented in Program 3.

We then compare the three following versions of the Cholesky application:

- the FIR version (Program 6) is closed to the SPMD code in Program 4 using a reduction operation;
- the FIP version (Program 7) is closed to Program 5 using a specific inspection phase and an asynchronous communication scheme;
- the MPI version refers to a hand-made high-optimized MPI version presented in [9].

The FIR and FIP versions use other optimizations not presented in this paper for handling irregular applications in HPF2. In both versions, the sparse matrix data structure is represented by using the *tree notation*, introduced in [4], based on the derived data type of Fortran 90. This notation avoids indirect data accesses coming from the standard irregular programming style, so that both compile-time and run-times techniques can be more easily performed. Both versions use the *processor subset* optimization in order to restrict the scope of communications associated with each iteration only to the required processor subset [4]. Finally, they also use a *task scheduling* optimization (SCHEDULE directive) where each iteration of the outer loop is associated with a task. Due to the irregularity property of the prefix operation, some tasks are independent from other tasks and can be executed in parallel (or in any order). The goal of our task scheduler is to order the task execution so as to minimize the global execution time while respecting the dependence constraints [6].

```
!HPF$ SCHEDULE (J = 1:K-1, bj = 1:NB(J), &
    ANY(A(J)%BCOL(bj)%BLOC) in A(K)%BCOL(1)%BLOC)

 DO K = 1, NB_BLOC_COL
!HPF$ ON HOME (A(J), J = 1:K-1, &
  ANY(A(J)%BCOL(:)%BLOC) in A(K)%BCOL(1)%BLOC) &
                                        , BEGIN
   TMP%BLOC%VAL = 0.0
!HPF$ INDEPENDENT, REDUCTION(TMP%BLOC%VAL)
   DO J = 1, K-1
      Update A(K)%BCOL with A(J)%BCOL    ! (BLAS)
   END DO
   A(K)%BCOL(:)%BLOC%VAL += TMP%BLOC%VAL
   LLt A(K)%BCOL(:)%BLOC%VAL   ! (LAPACK + BLAS)
!HPF$ END ON
 END DO
```

Program 6. FIR HPF Pseudo Code

```
!HPF$ PREFIX (A(:)%BCOL(:)%BLOC(:,:)%VAL)

!HPF$ SCHEDULE (J = 1:K-1, bj = 1:NB(J), &
    ANY(A(J)%BCOL(bj)%BLOC) in A(K)%BCOL(1)%BLOC)

 DO K = 1, NB_BLOC_COL
!HPF$ ON HOME (A(K)), BEGIN
   LLt A(K)%BCOL(:)%BLOC%VAL   ! (LAPACK + BLAS)
!HPF$ INDEPENDENT, PREFIX(A%BCOL%BLOC%VAL)
   DO J = K+1, NB_BLOC_COL
      Update A(J)%BCOL with A(K)%BCOL    ! (BLAS)
   END DO
!HPF$ END ON
 END DO
```

Program 7. FIP HPF Pseudo Code

Two libraries have been developed to support these optimizations. The T*ri*D*en*T library [4] supports distributed trees. The C*o*L*u*MBO library supports all the inspector/executor techniques that we have designed especially for irregular processor subsets, irregular loop index inspection [4], task scheduling [6] and irregular prefix operations.
We currently use these libraries by writing HPF2 codes with explicit calls to T*ri*D*en*T and C*o*L*u*MBO primitives. Then, we use the HPF compilation platform ADAPTOR [5] to obtain the final SPMD codes.

Matrix	C.B.	Col.	NNZ	NOp	Av. Coeff.
BCSSTK32	4286	44609	5.5 M	1.3 GFlop	0.368%
GRID 511	8216	261121	12 M	2.5 GFlop	0.121%
OILPAN	8024	73752	9.5 M	3.35 GFlop	0.129%
CUBE 31	1050	29791	8 M	5.5 GFlop	3.599%
CUBE 39	1129	59319	22 M	22 GFlop	4.518%

Figure 3. Properties of different sparse matrices

Experiments have been performed on IBM SP2 with 16 processors on various sparse matrices whose characteristics are described in Figure 3 (the number of Column-Blocks (C.B.), of Columns (Col.), of non zero elements (NNZ) and of operations (NOp)). The last column gives the average coefficient of irregularity. The average coefficient means that each column-block `I` has (on average since each column-block has its own subset B_i) `I` $\times c$ left-column-blocks which contribute to its irregular prefix operation. Matrices are distributed according to the *subtree-to-subcube* mapping [7] using an `INDIRECT` format; this mapping leads to an efficient reduction of communication while keeping a good load balance between processors.

Figure 4(a) gives the execution times obtained in seconds (for `FIR` and `FIP` versions, for global (inspection+execution) time and for execution time). The `FIP` versions are clearly better than the `FIR` ones (the gain varies from 3 to 62% for global time and from 0 to 35% for execution time). This great improvement comes from the asynchronous communication scheme, which allows communication/computation overlap. As noted in our basic experiments (cf. section 3.1), we can see that the lower the coefficient of irregularity, the higher the gain of the `FIP` version with regard to the `FIR` version. We can also note that the inspector time is proportional to the number of column-blocks (in accordance with the inspector SPMD code) and to the number of elements in B_i.

Finally, Figure 4(b) shows the time ratio of the `FIR` and `FIP` versions with regard to the MPI version. We can see that the ratio between the global time and the MPI time grows as the cost of the inspector time (from 1.2 for CUBE 31 to 3.5 for BCSSTK 32). However, when we consider only the execution time, the ratio is only about 1.11 to 1.18, except for the BCSSTK 32 (1.42, probably not enough work for a good scalability) for `FIP` and about 1.16 to 1.84 for `FIR`.

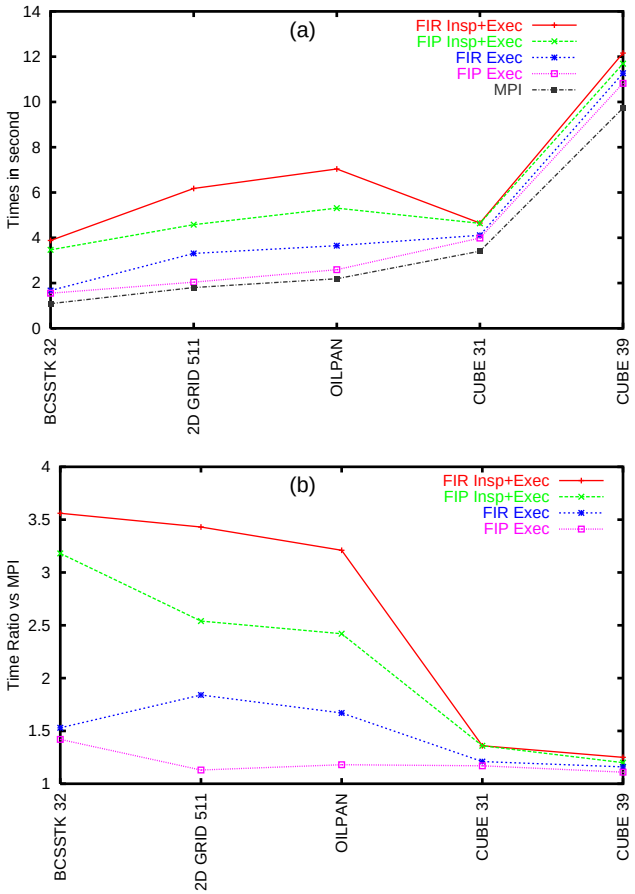

Figure 4. Times (a) and Time ratio (b) of Cholesky factorizations on different matrices on 16 processors

Ratio between regular HPF code and hand-made MPI code is generally about 1.50, and is worst in irregular application. So the results presented here show the great interest of our approach to obtain efficient irregular codes based on irregular prefix operations. This approach combines inspector/executor techniques for processor subset, task scheduling and asynchronous prefix operations.

4. Conclusion

The new `PREFIX` clause and directive added to the HPF2 language with an appropriate inspector/executor mechanism enable an efficient implementation of *progressive irregular prefix operations*. The proposed execution optimizations are based on an asynchronous communication scheme and communication/computation overlap. The C*o*L*u*MBO library is our run-time support for this inspector/executor. The experimental results achieved on basic examples and on a sparse Cholesky factorization applied on real size problems show the great interest of our approach.

Our future work is to study some other kinds of irregu-

lar computations such as unbalanced computations, and to extend our approach to the new SMP architecture, which involves both message passing and shared memory programming styles. We have implemented a prototype source-to-source compiler, not shown in this paper, which is able to translate simple HPF2 codes with our new directives and clauses into HPF2 codes with the corresponding inspection/execution code. So, we also study the possibility to transfer our knowledge of its implementation into a real HPF compiler in order to allow the compilation of more complicated HPF2 source codes.

References

[1] C. Ashcraft, S. Eisenstat, J.-H. Liu, and A. Sherman. A Comparison of Three Column Based Distributed Sparse Factorization Schemes. In *Fifth SIAM Conference on Parallel Processing for Scientific Computing*, 1991.

[2] P. Banerjee, J. Chandy, M. Gupta, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The PARADIGM compiler for distributed-memory message passing multicomputers. In *The First International Workshop on Parallel Processing*, pages 322–330, Bangalore, India, Dec. 1994.

[3] S. Benkner. HPF+: High Performance Fortran for advanced scientific and engineering applications. *Future Generation Computer Systems*, 15(3):381–391, 1999. also in Tech. Report TR 99-1 from *Institute for Software Technology and Parallel Systems, University of Vienna*, with E. Laure and H. Zima.

[4] T. Brandes, F. Brégier, M.-C. Counilh, and J. Roman. Contribution to Better Handling of Irregular Problems in HPF2. In *Proceedings of EURO-PAR'98*, volume 1470 of *LNCS*, pages 639–649, Southampton, UK, Sept. 1998. Springer-Verlag. Also available as a LaBRI Research Report RR 120598, 1998.

[5] T. Brandes and F. Zimmermann. ADAPTOR — A transformation tool for HPF programs. In K. M. Decker and R. M. Rehmann, editors, *Programming environments for massively parallel distributed systems: working conference of the IFIP WG10.3, Ascona, Italy*, pages 91–96, Cambridge, MA, USA, Apr. 25–29 1994. Birkhauser Boston Inc.

[6] F. Brégier, M.-C. Counilh, and J. Roman. Scheduling loops with partial loop-carried dependencies. Submitted to Special Issue on "Parallel Computing for Irregular Applications", 2000.

[7] K. G. et al. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, 1990.

[8] A. George and J.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[9] P. Henon, P. Ramet, and J. Roman. A Mapping and Scheduling Algorithm for Parallel Sparse Fan-In Numerical Factorization. In LNCS, editor, *Europar'99 Parallel Processing*, number 1685, pages 1059–1067, Toulouse, France, Aug. 1999.

[10] HPF Forum. *High Performance Fortran Language Specification*, Jan. 1997. Version 2.0.

[11] Y.-S. Hwang, B. Moon, S. Sharma, R. Ponnusamy, R. Das, and J. Saltz. Runtime and Language Support for Compiling Adaptive Irregular Programs on Distributed Memory Machines. *Software - Practice and Experience*, 25(6):597–621, 1995.

[12] A. Lain. *Compiler and Run-time Support for Irregular Computations*. PhD thesis, University of Illinois, 1996.

[13] A. Lain and P. Banerjee. Exploiting spatial regularity in irregular iterative applications. In *Proceedings of the 9th International Symposium on Parallel Processing (IPPS'95)*, pages 820–826, Los Alamitos, CA, USA, Apr. 1995. IEEE Computer Society Press.

[14] S. Mukherjee, S. Sharma, M. Hill, J. Larus, A. Rogers, and J. Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 68–79, July 1995.

[15] R. Ponnusamy, Y. Hwang, R. Das, J. Saltz, A. Choudhary, and G. Fox. Supporting Irregular Distributions in Fortran 90D/HPF Compilers. *IEEE Parallel and Distributed Technology*, 1995. Technical Report CS-TR-3268 and UMIACS-TR-94-57.

[16] J. Wu, R. Das, J. Saltz, H. Berryman, and S. Hiranandani. Distributed Memory Compiler Design for Sparse Problems. *IEEE Transactions on Computers*, 44(6):737–754, 1995.