Contribution to Better Handling of Irregular Problems in HPF2

Thomas Brandes, GMD/SCAI

Frédéric Brégier, Marie Christine Counilh,

Jean Roman, LaBRI

HUG'98 June 1998

bregier@labri.u-bordeaux.fr

Plan

- Irregularity and HPF2
- Related Work
- Our Approach
- Experimentations

Irregularity and HPF2

- HPF is well suited for regular applications
- **Irregularity:**

```
!HPF$ DISTRIBUTE (BLOCK) ::  X, Y, IX, IY
    DO I = 1, N
        X(I) = Y(IX(I)) - Y(IY(I))
    END DO
```

  - Indirection only known at run-time
  - Data distribution often irregular (`GEN_BLOCK`, `INDIRECT`)
  - Non expressible alignment
- **Problems at compile-time:**
  - Dependencies not known at compile-time
  - Prevents regular standard optimizations: extraction of communications, loop optimization...

Related Work

- **Compilation Techniques:** Help the compiler to handle irregularity better. Often limited to sparse matrices.

  1. "Sparse Compiler" by Bik and Wijshoff: dense matrix code with its topology
     Limited to sparse matrix with known topology
  2. Vienna Fortran: sparse matrix code specifying the sparse matrix structure used:

     REAL A(NA,NB), SPARSE (CCS(AD,AC,AR)), DYNAMIC

     Limited to sparse matrix structures known by the compiler

- **Run-Time at Compile-Time Techniques:** Optimized execution by inspection/execution. Several libraries: CHAOS, PARTI, PILAR, RAPID

  Essentially iterative computations (where inspection cost covered by gain of multiple executions)

Our Approach

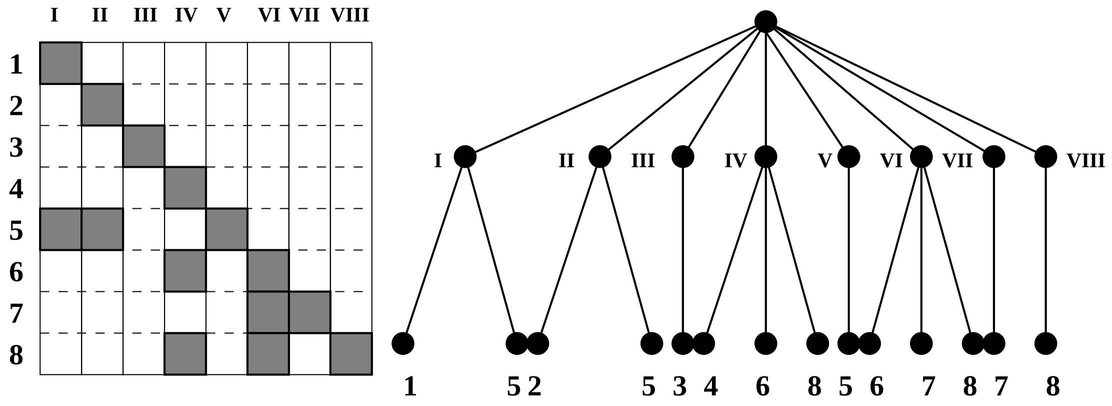
1. Offer a programming style close to that of the regular case with a **generic** data structure, to allow for the standard compile-time optimizations

2. Enable more specific inspection/execution for handling of irregularity in HPF2

3. Validate by experimentation with the TriDenT library

Generic Data Structure: the Tree

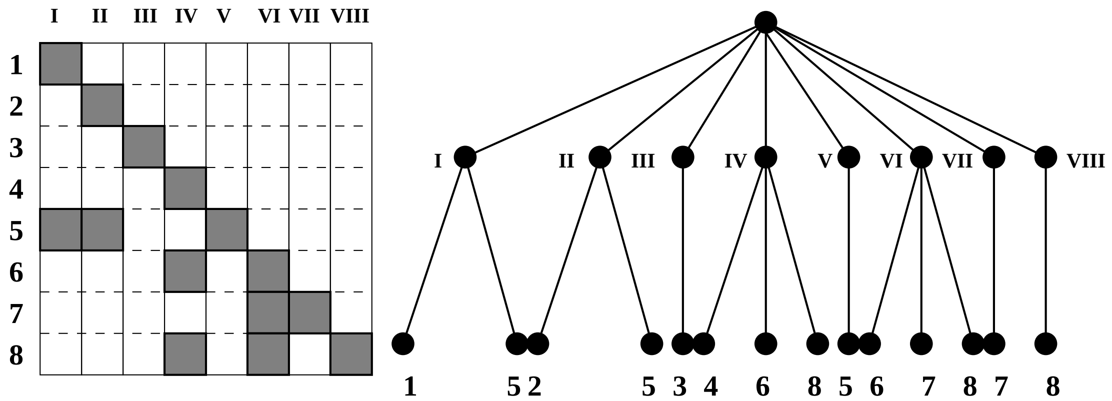
All data structures with hierarchical access can be represented as trees.
For instance, a column access oriented sparse matrix:



- The i^{th} element of A represents the i^{th} column
- The j^{th} element of A(i) is the j^{th} non zero element of the i^{th} column
- The first level of this tree represents the column sets of the sparse matrix
- The second level represents row numbers and associated values

The Tree: Representation in HPF2

- Derived Data Type of Fortran 90 ($\in$ HPF2)




```
type level2
     integer ROW
     real VAL
end type level2

type level1
     type (level2), pointer ::  COL(:)
end type level1

type (level1), allocatable ::  A(:)
!ADP$ TREE A
```

- Analogy between the level notion for trees and the classical dimension notion for arrays

  `A(i)%COL(j)%VAL` $\equiv$ `A(i,j)` *with a dense matrix*

  `A(i)%COL(:)%VAL` $\equiv$ `A(i,:)`

- The new `TREE` directive ensures some restrictions ($\notin$ HPF2):
  - No recursions, no pointer target assignment(allocatable)

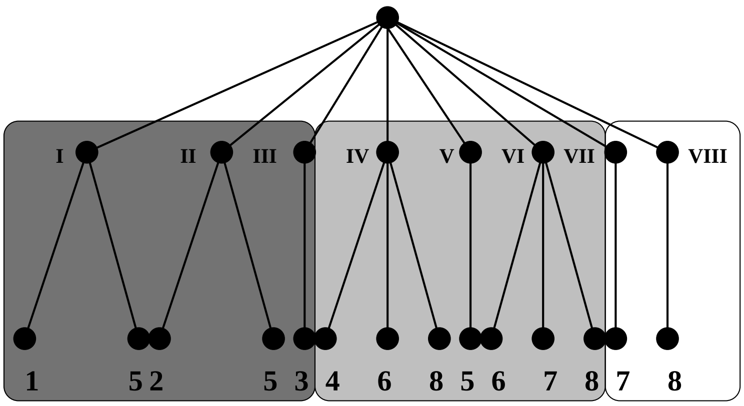
- Depth known at compile-time
- no more than one subtree by node

The Tree: Distribution in HPF2

- Distribution from only one level (*reference level*) (approved extensions of HPF2)

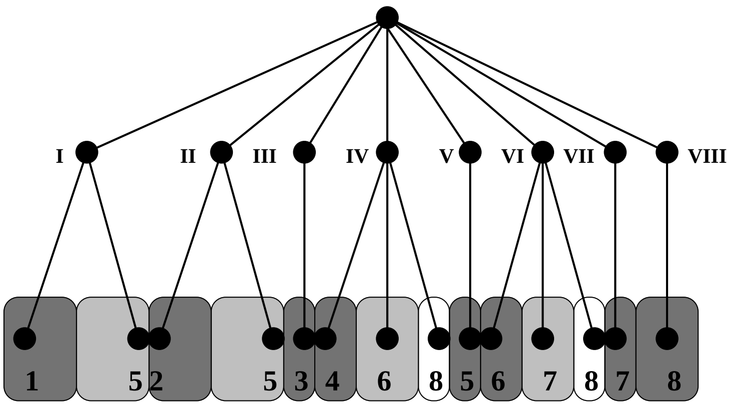
- The preceding levels (upper or parent) are replicated

- The following levels are distributed according to the distribution of the reference level



BLOCK on the reference level 1

```
type level2
     integer ROW
     real VAL
end type level2

type level1
     type (level2), pointer ::  COL(:)
end type level1

type (level1), allocatable ::  A(:)
!ADP$ TREE A
!HPF$ DISTRIBUTE A(BLOCK) !(level 1)
```



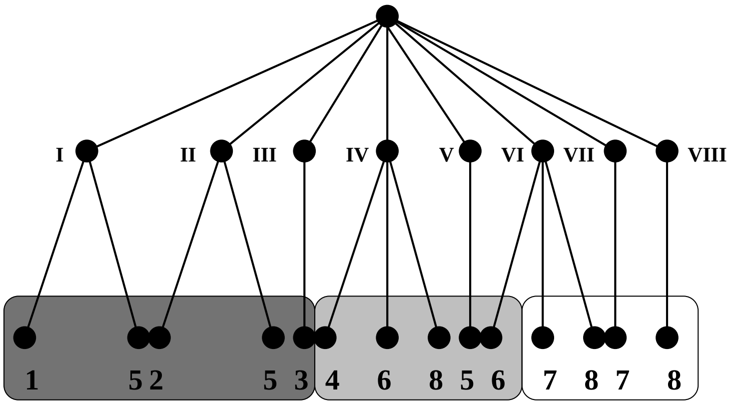
BLOCK on the reference level 2

```
type level2
     integer ROW
     real VAL
end type level2

type level1
     type (level2), pointer ::  COL(:)
!HPF$ DISTRIBUTE COL(BLOCK) !(level 2)
end type level1

type (level1), allocatable ::  A(:)
!ADP$ TREE A
```

The Tree: Extended Distribution

Proposed extension: distribute a tree along a whole level

```
type level2
     integer ROW
     real VAL
end type level2

type level1
     type (level2), pointer ::  COL(:)
end type level1

type (level1), allocatable ::  A(:)
!ADP$ TREE A
!ADP$ DISTRIBUTE TREE A(*,BLOCK) !(HPF2/TREE)
```



BLOCK on level 2 (HPF2/TREE)

The Tree: Advantages

- Data type adapted to the algorithm (according to the access and the distribution, generic data structure)
- Remove indirections from the standard irregular codes (for example, CCS format)

  `A(i)%COL(j)%VAL` $\equiv$ `A_VAL(A_COL(i)+j)`

  `A(i)%COL(:)%VAL` $\equiv$ `A_VAL(A_COL(i)+1 : A_COL(i+1))`
- Ensure the logical alignment (without explicit `ALIGN`), known by the compiler, between the different tree components
- Standard regular optimizations by the compiler (dependence analysis, data locality)
- Derived Data Type $\in$ HPF2 (and approved extensions): portability (even without the TREE directive)

The Tree: Disadvantages

- Must (re)write the code with this particular data structure
- No (or few) commercial compiler accepting derived data types and approved extensions (Digital Fortran/HPF Compiler)
- Some experimental compilers do (VFCS, HPF+, PGI Development Compiler, Adaptor soon)

Run-Time Optimizations for Irregularity in HPF2

- HPF2: `ON` directive.

```
    DO K = 1, N
!HPF$ ON HOME(A(B(K)%COL(:)%DEP)) BEGIN
        Cholesky on column K
!HPF$ END ON
!  B(K)%COL(:)%DEP ≡ left dependent columns for K
    END DO
```

- **Irregular active processor sets** for the instructions between `BEGIN/END ON`.

- For loop with **loop carried dependencies** known at run-time but **computable before the entry of the loop**
  - compute processor sets in advance
    $\Rightarrow$ avoids synchronizations at run-time
  - Example: Cholesky

- **Set Inspector**.

```
!HPF$ INDEPENDENT
    DO K = 1, N
!HPF$ ON HOME(B(K)) BEGIN
        DO J = 1, SIZE(B(K)%COL)
            I = B(K)%COL(J)%DEP
            ADD PROC(A(I)) TO ACTIVE_PROC(K)
        END DO
!HPF$ END ON
    END DO
    FINALIZE SET
```

Run-Time Optimizations

- In order to allow easier writing and more efficient inspection, proposed extension for the `HOME` clause (`FORALL` syntax):

```
    DO K = 1, N
!HPF$ ON HOME(A(J), J=1:K, ANY(A(J)%COL(:)%ROW.EQ.K))
        Cholesky on column K
!HPF$ END ON
    END DO
```

- Set Inspector becomes:

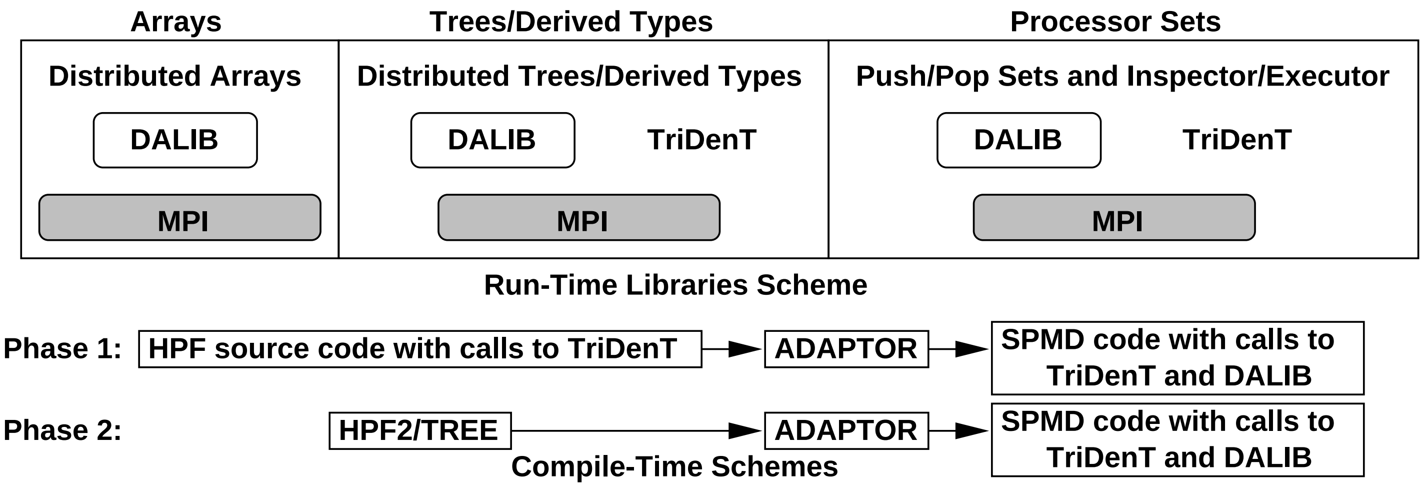
```
!HPF$ INDEPENDENT
    DO J = 1, N
!HPF$ ON HOME(A(J)) BEGIN
        PJ = PROC(A(J))
        DO I = 1, SIZE(A(J)%COL)
            ADD PJ TO ACTIVE_PROC(A(J)%COL(I)%ROW)
        END DO
!HPF$ END ON
    END DO
    FINALIZE SET
```

- Loop Index Inspector
- Communications Inspector: extractable or across-iteration

Realization: TriDenT

1. Implementation of the trees (no compiler supporting derived types)
2. Implementation of the inspectors/executors

- First purpose: allow the writing of an HPF code with explicit calls to TriDenT primitives *(Phase 1)*
- Designed to be integrated in the compilation platform **ADAPTOR** *(Phase 2)*

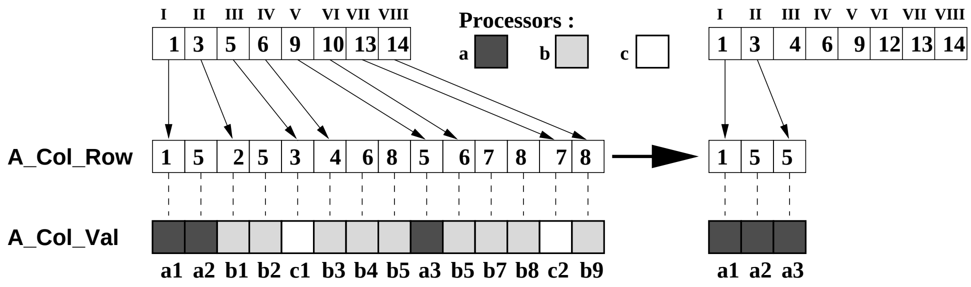


TriDenT

1. The Trees:

   - Implementation = one-dimensional array for each component of the associated derived types
   - All HPF2 distributions supported, translated in `GEN_BLOCK` format

     `!HPF$ DISTRIBUTE A(INDIRECT(/a,b,c,b,a,b,c,b/))`



   - Efficient access functions (tree and subtree)
   - Some implementation restrictions:
     - No re-allocation nor re-distribution
     - Only one-dimensional arrays in derived types

2. Inspectors/Executors:

   - Main goal: to be efficient even if the executor is used only once
   - 3 phases: local, global, execution

Experimental Results

- Cholesky oriented column factorization for sparse matrices
- Matrix: 255x255 2D-grid finite element problem, with re-ordering and symbolic factorization (matrix L)
- 65025 columns, $<$ 0,02% non-zeros after fill-in
- Tree structure: same as above
- `INDIRECT` distribution according to a "subtree-to-subcube" mapping (increase parallelism)
- Tested on a IBM SP2 with 16 processors

Version	ON HOME (T)ree `B` or (E)xtended Clause	Indices	Communications
0	-	No	No
A	T	No	No
B	E	Yes	No
C	E	Yes	Yes

- **Effectiveness of the basic Set inspection**: magnitude of 10 between version **0** (without Set Inspector) and version `A` (450s / 36s on 16 processors)

HPF2/TREE Sparse Cholesky Fan In

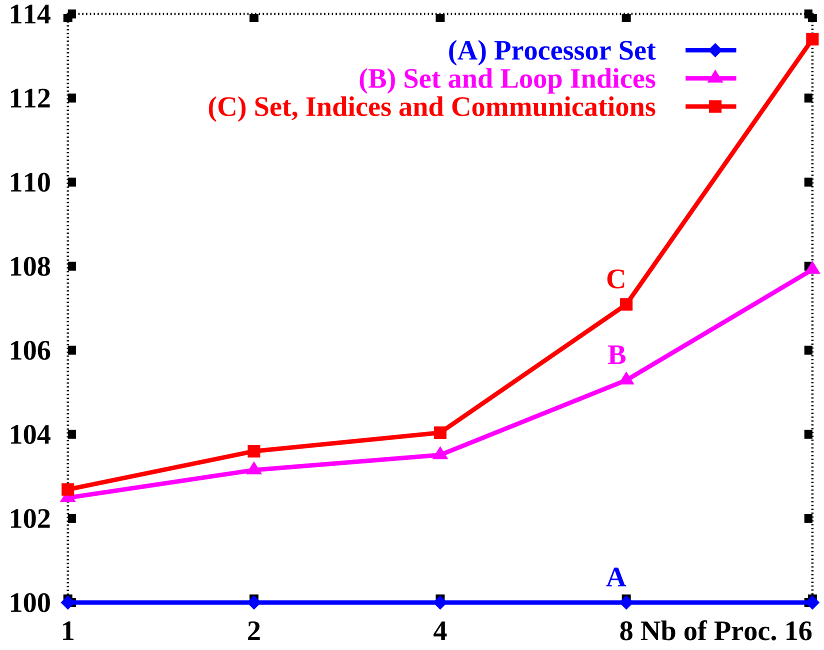
```
    DO k = 1, n

       DO j = 1, k-1
!HPF$ ON HOME (A(j), ANY(A(j)%COL(:)%ROW .EQ. k))
          CMOD(val, A(k)%COL(:)%ROW, A(j)%COL(:)%VAL, &
               A(j)%COL(:)%ROW)
       !val(:)  = val(:)  - A(j)%COL(:)%VAL * A(j)%COL(k)%VAL
       END DO

    END DO
```

```
        allocate (val(size(A(k)%COL)))
        val(:)  = 0.0

!HPF$ INDEPENDENT, REDUCTION(val)



        A(k)%COL(:)%VAL = A(k)%COL(:)%VAL + val(:)

        A(k)%COL(1)%VAL = sqrt(A(k)%COL(1)%VAL)
        CDIV(A(K)%COL(:)%VAL)
        !A(k)%COL(:)%VAL = A(k)%COL(:)%VAL / A(k)%COL(1)%VAL
```

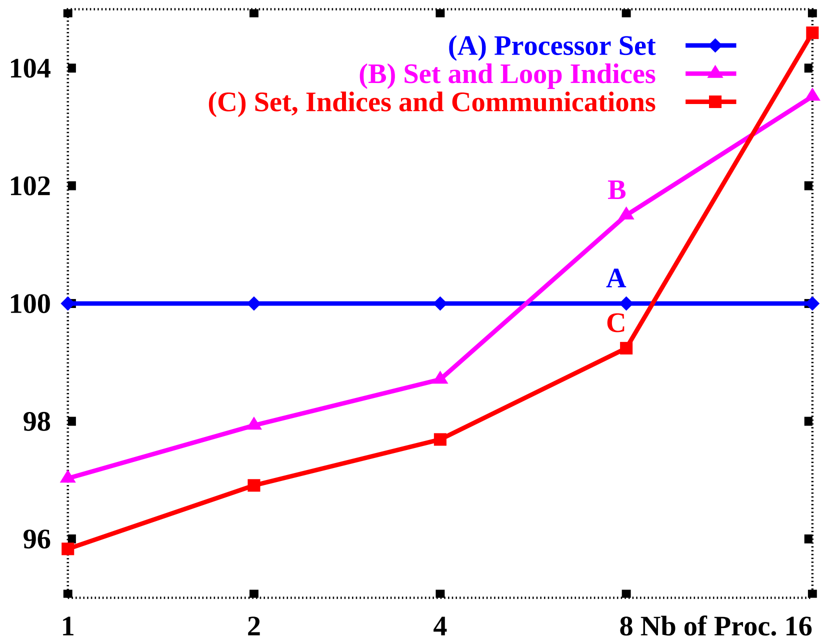
```
!HPF$ DISTRIBUTE A(INDIRECT(map))
```

```
!HPF$ ON HOME (A(j), j = 1:k, ANY(A(j)%COL(:)%ROW .EQ. k)), &
              !  NEW(val), BEGIN
```

```
!HPF$ END ON
```

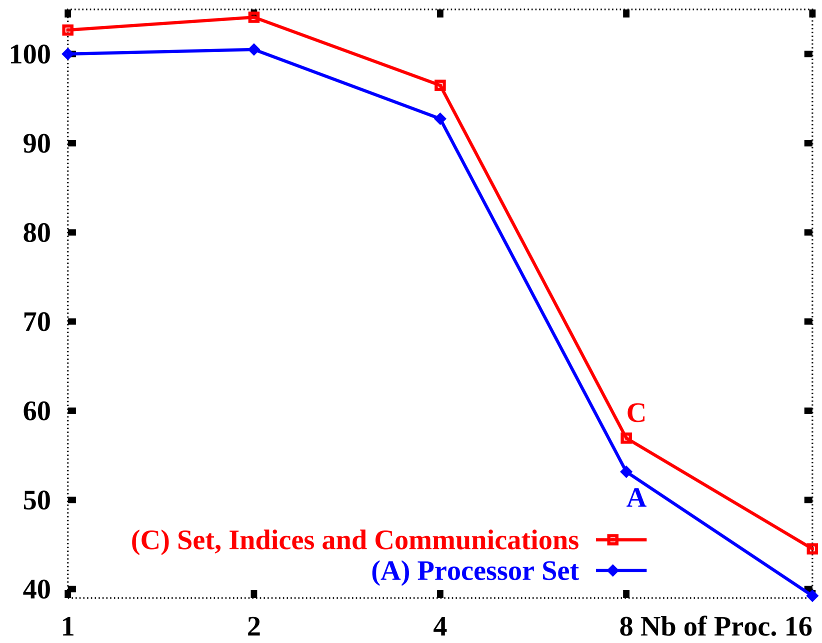


Relative Efficiencies (Execution Time Only) Compared to **A**

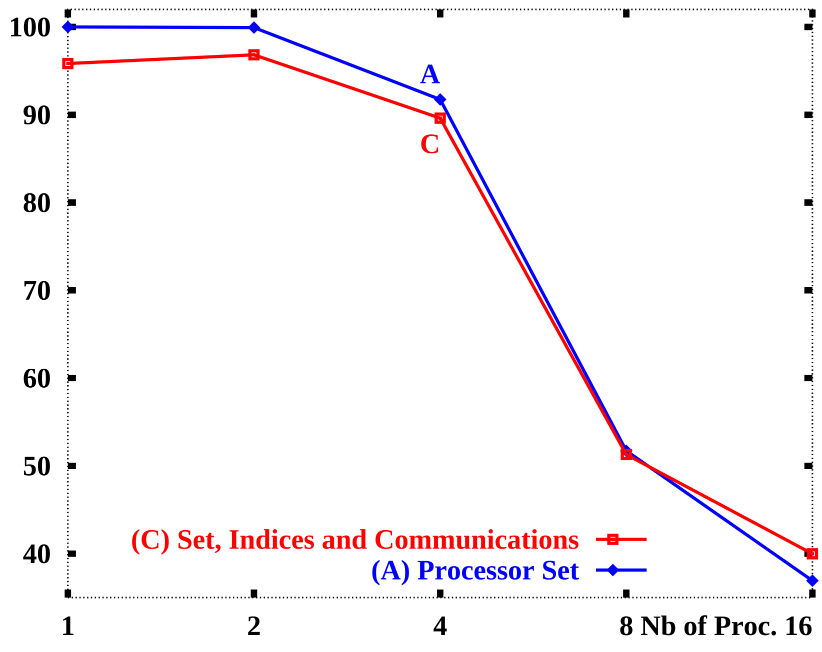


Relative Efficiencies (Global Time) Compared to **A**

Inspection costs repayed by a single executor gain for more than 8 processors



Relative Efficiencies (Execution Time Only) of **A, C** vs **A** on 1 Proc.



Relative Efficiencies (Global Time) of **A, C** vs **A** on 1 Proc.

Reasonable Efficiency: about 90% on 4 processors and 40% on 16 processors

Conclusion

- Writing irregular codes with the Trees (with hierarchical access data structure, not limited to sparse matrix!)
  $\Rightarrow$ more information given to the compiler
- Better analysis, so standard regular optimizations by the compiler
- `ON` directive proposed extension: automatic irregular set creation and specific inspection/execution techniques.

Perspectives

- Extension of the tree properties - TriDenT -: geometrical distributions , flexibility (re-allocation and re-distribution).
- Extension of the inspectors/executors: task parallelism induced by the loops, especially using threads.
- Integration of TriDenT in the compilation platform ADAPTOR.

Experimentations: HPF2/TriDenT Code

```
!HPF$ INDEPENDENT
    DO j = 1, n
        Add Proc (A(j)) to Active_Proc (A(j)%COL(1)%ROW)
        DO i = 2, size(A(j)%COL)
            Add Proc (A(j)) to Active_Proc (A(j)%COL(i)%ROW)
            Add Index i to Loop (A(j)%COL(i)%ROW)
        END DO
    END DO
    Finalize Group & Loop
!HPF$ INDEPENDENT
    DO k = 1, n
        Add A(k)%COL(:)%ROW to EXTR_COMM (k) GROUP(k)
    END DO
    Finalize Comm

    DO k = 1, n
        Push Active_Proc(k)
        allocate (val(size(A(k)%COL)))
        val(:)  = 0.0
        allocate (row(size(A(k)%COL)))
        Get_Comm(row, A(k)%COL(:)%ROW)
!HPF$ INDEPENDENT, REDUCTION(val)
        DO j1 = 1, size(Loop(k))
            j = Loop(k,j1)
            CMOD(val, row, A(j)%COL(:)%VAL, A(j)%COL(:)%ROW)
        END DO
        A(k)%COL(:)%VAL = A(k)%COL(:)%VAL + val(:)
        CDIV(A(K)%COL(:)%VAL)
        Pop Active_Proc(k)
    END DO
```