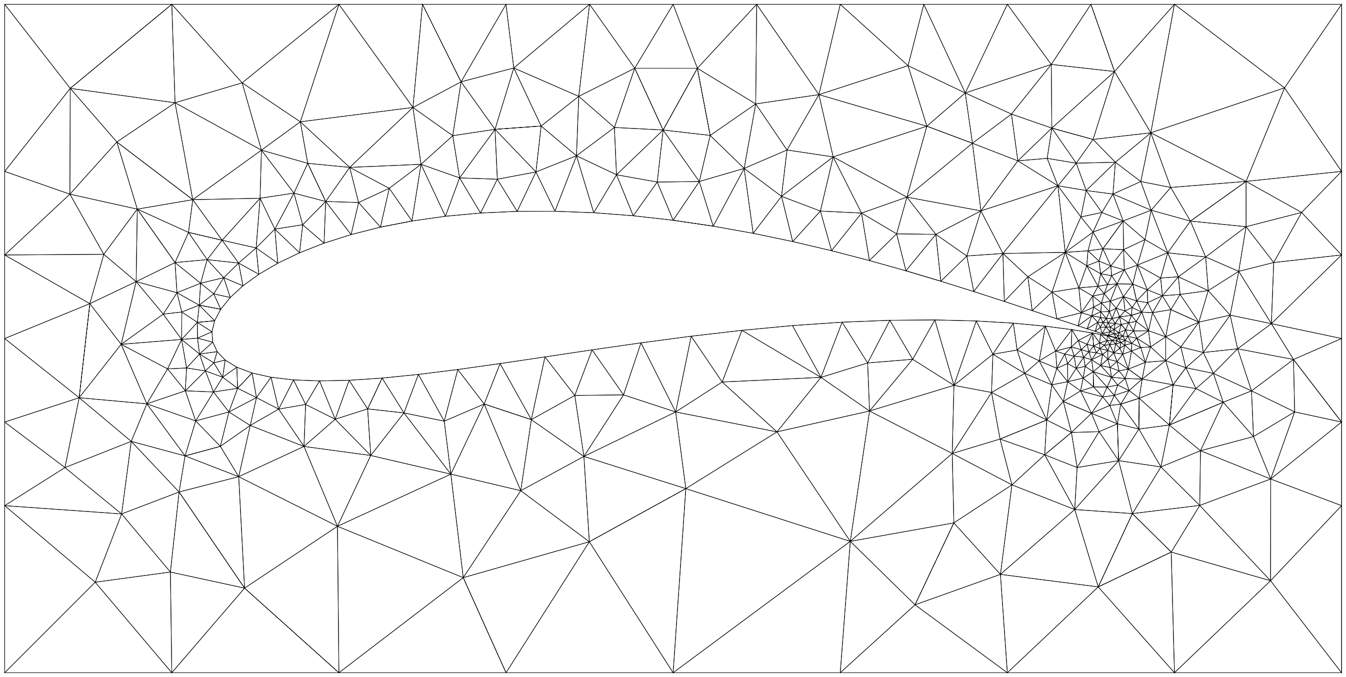
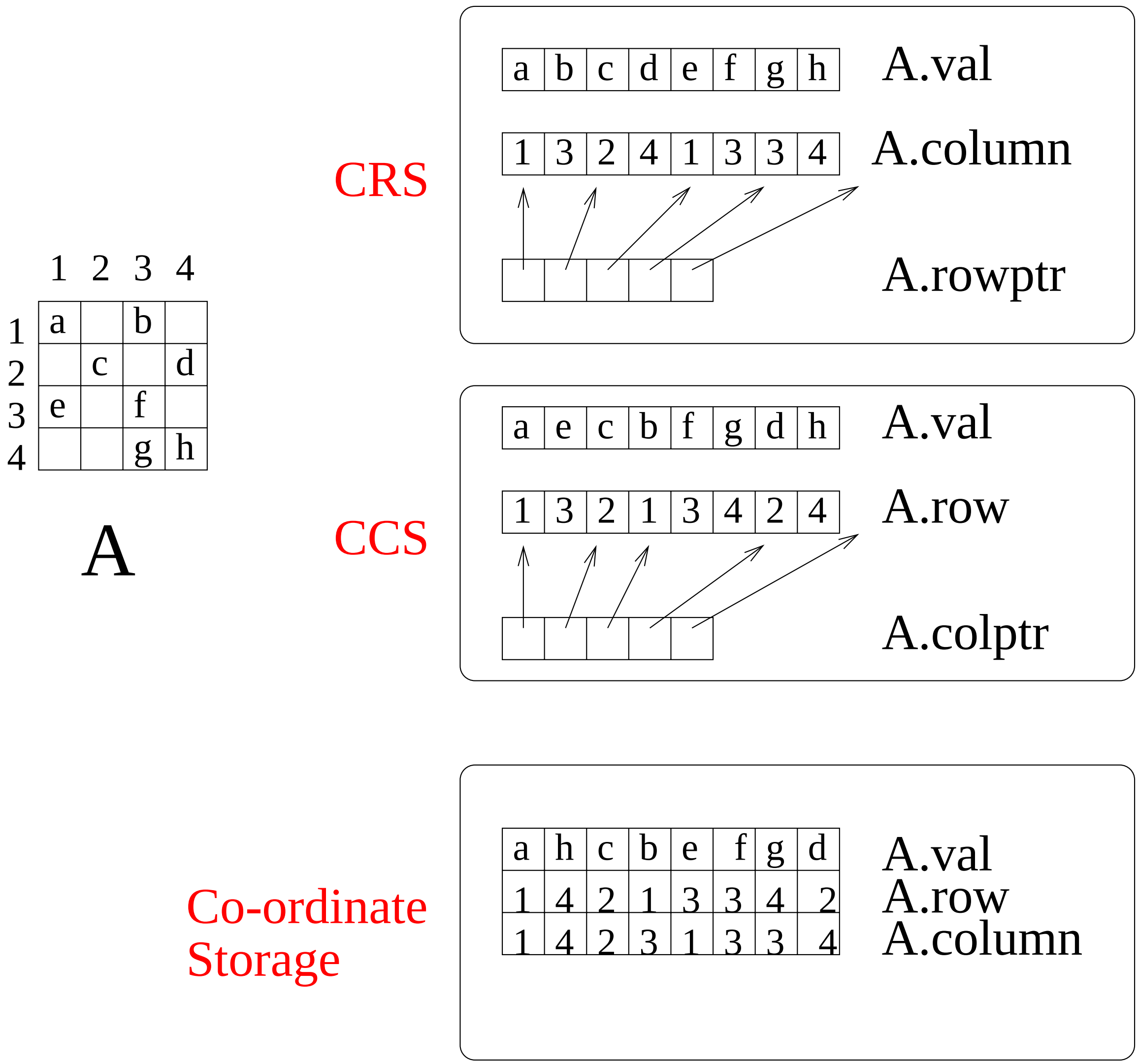
Sparse Compilation

Motivation for Sparse Codes



- Consider ¤ux, heat, or stresses – interactions are between neighbors.
- Linear equations are sparse. Therefore, matrices are sparse.

Three Sparse Matrix Representations



Indexed access to a row

Indexed access to a column

Indexed access to neither rows nor columns

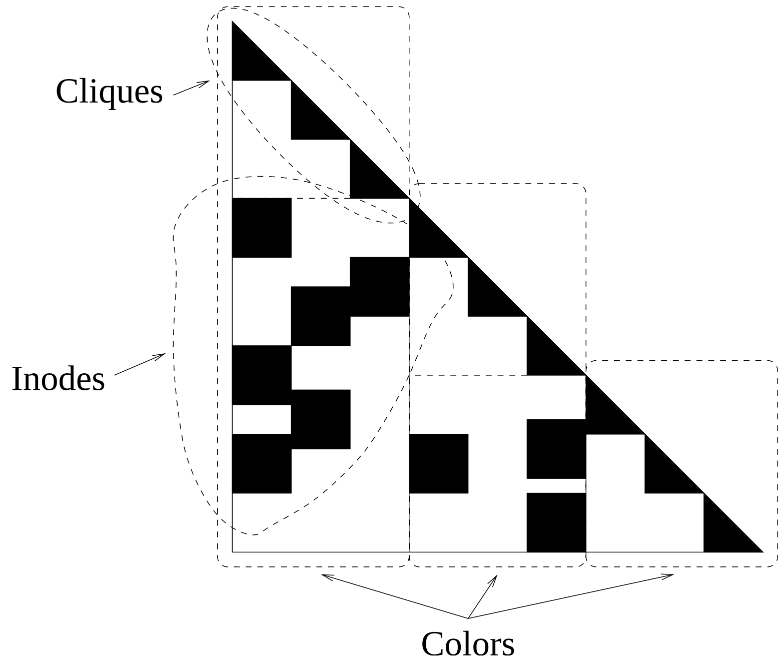
Jagged Diagonal (JDiag)

$$
\begin{array}{c}
\\ 1 \\ 2 \\ 3 \\ 4
\end{array}
\begin{array}{c}
\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\begin{pmatrix}
a & & b & \\
& c & d & e \\
f & g & & \\
& & & h
\end{pmatrix}
\end{array}
\longrightarrow
\begin{array}{c}
\\ 2 \\ 1 \\ 3 \\ 4
\end{array}
\begin{array}{c}
\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\begin{pmatrix}
& c & d & e \\
a & & b & \\
f & g & & \\
& & & h
\end{pmatrix}
\end{array}
\longrightarrow
\begin{bmatrix}
c & d & e \\
a & b & \\
f & g & \\
h & &
\end{bmatrix}
$$

perm	2	1	3	4				
adiagptr	1	5	8	9				
acolind	2	1	1	4	3	2	2	4
avalues	c	a	f	h	d	b	g	e

- Long “vectors”
- Direction of access is not row or column

BlockSolve (BS)



- Dense submatrices
- Colors $\longrightarrow$ cliques $\longrightarrow$ inodes.
- Composition of two storage formats.

The effect of sparse storage

Name	N	Nzs	Diag.	Coor.	CRS	JDiag	B.S.
nos6	675	3255	$\boxed{38.658}$	5.441	20.634	32.945	2.570
$2 \times 25 \times 1$	625	3025	$\boxed{37.907}$	5.650	21.416	32.952	2.593
nos7	729	4617	$\boxed{35.749}$	4.836	20.000	27.830	3.259
$2 \times 10 \times 3$	300	4140	27.006	9.359	29.881	$\boxed{33.727}$	17.457
medium	181	2245	23.192	7.888	29.874	$\boxed{32.583}$	19.633
bcsstm27	1224	56k	15.130	4.807	23.677	21.604	$\boxed{28.907}$
e05r0000	236	5856	8.534	4.841	$\boxed{26.642}$	25.085	SEGV
$3 \times 17 \times 7$	34.4k	1.6M	8.478	4.752	23.499	11.805	$\boxed{27.615}$

NIST Sparse BLAS

- Algorithms

  1. Matrix-matrix products (MM),

     $$C \leftarrow \alpha AB + \beta C \qquad C \leftarrow \alpha A^T B + \beta C,$$

     where A is sparse, B and C are dense, and α and β are scalars.

  2. Triangular solves,

     $$C \leftarrow \alpha D A^{-1} B + \beta C \qquad C \leftarrow \alpha D A^{-T} B + \beta C$$
     $$C \leftarrow \alpha A^{-1} D B + \beta C \qquad C \leftarrow \alpha A^{-T} D B + \beta C$$

     where D is a “(block) diagonal” matrix.

  3. Right permutation of a sparse matrix in Jagged Diagonal format,

     $$A \longrightarrow AP \qquad A \longrightarrow AP^T$$

  4. Integrity check of sparse A.

NIST Sparse BLAS (cont).

- Storage formats
  - Point entry – each entry of the storage format is a single matrix element.
    - Coordinate
    - CCS
    - CRS
    - Sparse diagonal
    - ITPACK/ELLPACK
    - Jagged diagonal
    - Skyline
  - Block entry – each "entry" is a dense block of matrix elements.
    - Block coordinate
    - Block CCS
    - Block CRS
    - Block sparse diagonal
    - Block ITPACK/ELLPACK
    - Variable Block compressed Row storage (VBR)

NIST Sparse BLAS (cont).

- Limitations
  - Huge number of routines to implement.

    User-level Only 4 routines.

    Toolkit-level 52 (= 4 routines * 13 formats) routines.

    Lite-level 2,964 (= 228 routines * 13 formats) routines.
  - Algorithms are not complete.

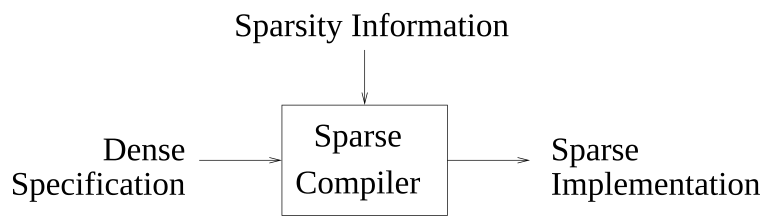
    E.g., Matrix assembly, Incomplete and complete factorizations.
  - Data structures are not complete.

    E.g., BlockSolve
  - Only one operand can be sparse.

    No sparse $C = A * B$.

A Sparse Compiler

- Still need to develop sparse applications.
- Want to automate the task.



Design goals:

- Programmer selects the sparse storage formats.
- Programmer can specify novel storage formats.
- Sparse implementations as ef£cient as possible.

Challenges for sparse compilation

- Describing sparse matrix formats to the compiler.
- Transforming loops for ef£cient access of sparse matrices.
- Dealing with redundent dimensions.
- Accessing only Non-zeros.

Describing Storage Formats – Random Access

- $A[i, j]$.
- Sparse matrices as objects with `get` and `set` methods.
- Dependencies are preserved.

```
for i = 1, n
    for j = 1, n
        y[i] += A.get(i,j) * x[j]
```

- inef£cient
  - Searching is inef£cient.
  - Useless compuation when $A[i, j] = 0$.

Describing Storage Formats – Sequential Access

- Stream of non-zeros, $< i, j, v >$.
- Sparse matrices as containers with iterations.

```
for <i,j,v> in A.nzs()
    y[i] += v * x[j]
```

- What about dependencies? Must know order of iteration.
- Simultaneous enumeration...

Describing Storage Formats – Sequential Access (cont.)

- Consider $C = A * B$, where A and B are sparse.
- With sequential access,

```
for <i,k,Av> in A.nzs()
    for <k',j,Bv> in B.nzs()
        if k = k' then
            C[i,j] += Av * Bv
```

- a better solution,

```
for <i,k,Av> in A.nzs()
    for <j,Bv> in B[k,:].nzs()
        C[i,j] += Av * Bv
```

- CRS gives us this type of access.

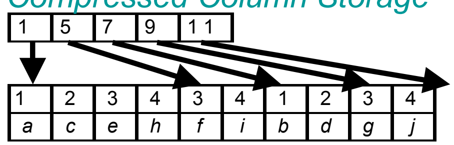
Indexed-sequential access

- Storage formats have heirarchical structure.
- Algebraic description of this structure.

- Nesting

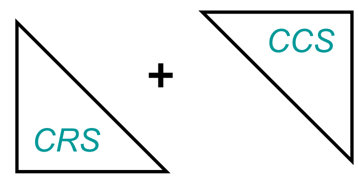
  $c \rightarrow r \rightarrow v$

  Compressed Column Storage



- Aggregation

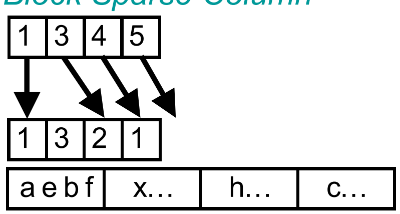
  $(r \rightarrow c \rightarrow v) \cup (c \rightarrow r \rightarrow v)$



- Linear Maps

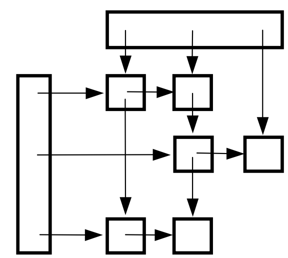
  $map\{br*B+or \leftrightarrow r,\ bc*B+oc \leftrightarrow c :$
  $bc \rightarrow br \rightarrow <or\ \ oc> \rightarrow v\}$

  Block Sparse Column



- Perspective

  $(r \rightarrow c \rightarrow v) \oplus (c \rightarrow r \rightarrow v)$



Conveying the structure to the compiler

- Annotations

```
!$SPARSE CRS: r -> c -> v
real A(0:99,0:99)
```

- Each production implemented as an abstract interface class

```
class CRS : public Indexed<int,
                             Indexed<int,
                                 Value<double> > >
```

Challenges for sparse compilation

$\surd$ Describing sparse matrix formats to the compiler.

$$r \longrightarrow c \longrightarrow v$$

- Transforming loops for ef£cient access of sparse matrices.
- Dealing with redundent dimensions.
- Accessing only Non-zeros.

Loop transformation framework

- Extend our framework for imperfectly nested loop
- For each statement – Statement space = (Iteration space, Data space)

```
for i =
    for j = ...
        S1: ... A[F1(i,j),F2(i,j)]
                      + B[G(i,j)] ...
```

  - Iteration space - as before
  - Data space - Product of sparse array dimensions,

$$S1 :< i, j, a_1, a_2, b >$$

- Product space - Cartesian product of statement spaces

Loop transformation framework (cont.)

- Finding embedding functions
  - Add constraints for array refs $a = Fi$.
  - Use Farkas Lemma, as before.
- Account for the structure of sparse matrices,
  - map – change of variables, $P' = TP$
  - perspective – choice
  - aggregation – make two copies of indices, $a \rightarrow a', a''$
- Transformations - *not tiling*, instead Data-centric
  - order the array indices, $a_1, a_2, b_1, b_2, \ldots$
  - Partial transformation, bring array indices outermost
  - Complete transformation.

Challenges for sparse compilation

$\surd$ Describing sparse matrix formats to the compiler.

$$r \longrightarrow c \longrightarrow v$$

$\surd$ Transforming loops for ef£cient access of sparse matrices.

- Augmented product space.
- Data-centric transformations.

- Dealing with redundent dimensions.
- Accessing only Non-zeros.

Redundent dimensions

- Dot product of two sparse vectors, $j \rightarrow v$.

```
for i = 1, n
    sum += R[i] * S[i]
```

- Statement and product space: $(i, r, s)^T$.
- Transform:$(r, s, i)^T$
- Constraints: $i = r = s$

```
for <ir,a> in R
    for <is,b> in S
        if ir = is then
            sum += a * b
```

- Two dimensions are redundent.
  - Dense code, random access: Replace s and i with r.
  - Sparse code, sequential access: simultanous enumeration

```
for <ir,a> in R,
        <is,b> in S,
        when ir=is
    sum += a * b
```

Connection with relational databases

- Relations – sets of tuples
- Join, $\bowtie$ – Constrained cross product.

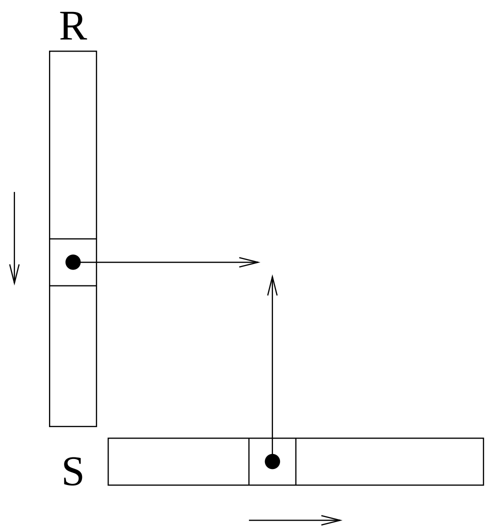
$$R \bowtie S = \{< i, a, b > \mid < i, a > \in R, < i, b > \in S\}$$

- Connection:
  - Sparse matrices as relations.
  - Simultaneous enumeration as $\bowtie$.

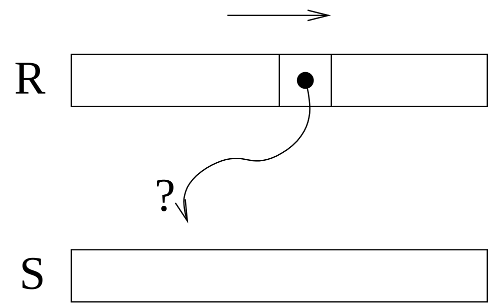
Join implementations

Implementations of $R \bowtie S$:

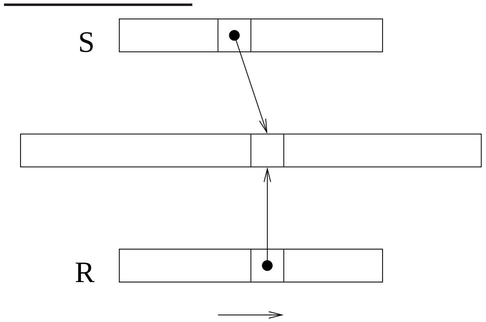
Nested loop join



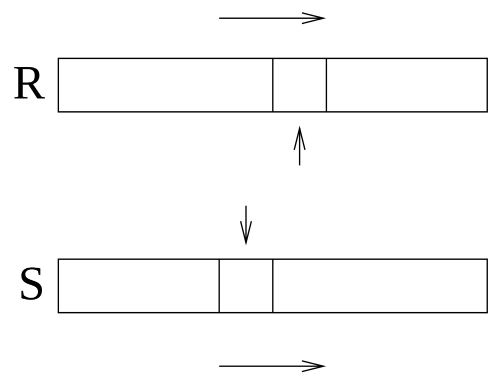
Index join



Hash join



Sort-merge join

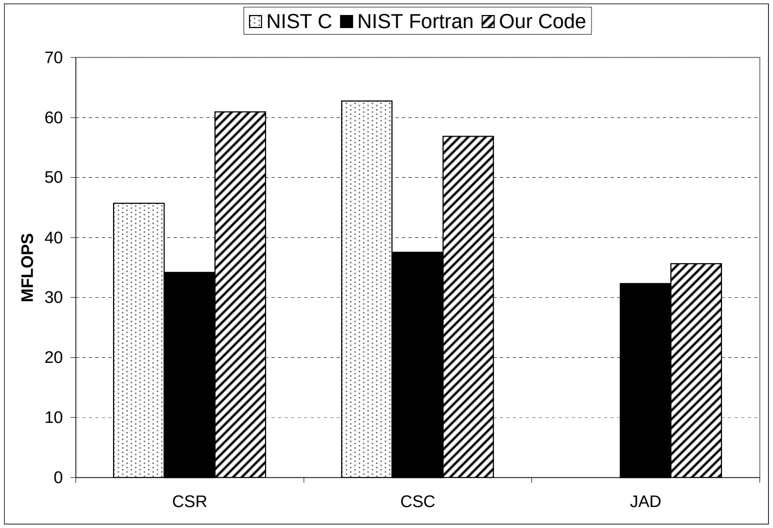


Simultaneous enumeration

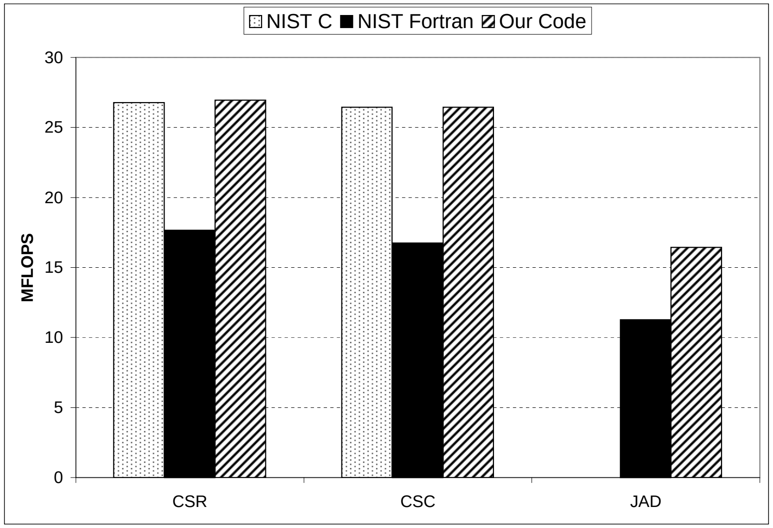
- Identifying the joins –
  - Let x be a vector of the transformed product space indices,
  - Let $Fx = f_0$ be the constraints on the indices (array access, ...),
  - Hermite Normal Form, $L = PFU$.
  - One join for each non-zero column of L.
- Af£ne constraints – more general join operation.
- Dependencies – constrain order of enumeration
- Checks for original loop bounds – use Fourier-Motzkin to simplify.

Results

Triangular solve - NIST C vs. NIST F77 vs. Bernoulli



SGI Octane, 300Mhz



Pentium II, 300Mhz