Parallelizing Sequential C Code Automatically

Contents of Lecture 9

- Reflections on explicit parallelism and OpenMP.
- Application classes needing automatic parallelization
- Loop-level parallelization using unimodular transformations
- The SUIF compiler
- Thread level speculative execution — a study of SPEC CPU2000 at Stanford

The Need for Automatic Parallelization

- There are huge amounts of source code which is sequential.
- We almost always want faster programs.
- Which classes of sequential software can be automatically parallelized?
- Why should we not always parallelize by hand?
- More precise question: how can we get sufficient performance...
  - ...at acceptable development cost and time?
  - ...for acceptable hardware costs?
- For the last 50 years or so, there has been a quest for automatically parallelizing sequential software.
- Should we write multi-threaded code or hope for automatic parallelization???

No Simple Answer

- Different projects have radically different needs!
- Different source codes are more or less easy to parallelize!
- There is no simple answer to the question whether automatic parallelization is worthwhile.
- If it solves your problem under various constraints, then it is worthwhile, as any other optimization.

Some Reflections on OpenMP

- Of the codes we have seen so far, for which can we realistically build tools which automatically can extract lots of parallelism?
- Automatically parallelizing sequential Java or C codes with complex recursive data structures (e.g. a graph)? Not likely!?
- Automatically rewriting sequential Scala programs and create actors? Not likely!?
- OpenMP loops? Yes, since such loops don't have dependences, at least there exist numerical loops which can be executed in parallel!!
- Since such loops exist — why do we need OpenMP??
- Because no compiler exists which is as reliable as a programmer inserting `#pragma parallel for` directives.
- However, with a sufficiently good parallelizing C compiler for numerical codes (i.e. matrix computations) we would not need OpenMP.
- In this lecture we will learn how close we are to such C compilers.

Safety of Parallelization

- The criterion to determine whether it's safe for a compiler to generate parallel code for a loop is, of course, whether the parallel code may produce different output.
- If different loop iterations access the same array element and at least one of the accesses is a store, then the loop must be executed sequentially.

From Simple to Hard Parallelization Problems

- As we will see, matrix computation codes where the loop bounds and array references are linear functions of the loop index variables are relatively easy to analyse.
- This analysis will then conclude either that the loop is sequential, or a certain number of outer loops can run in parallel.
- At the other end are codes which for example contain dynamically allocated recursive data structures, e.g. lists or trees — it's very difficult or impossible to determine for a compiler wether it is safe to execute parts of such codes in parallel.
- In between are codes which sometimes can be analysed using symbolic analysis, i.e. mathematical reasoning with symbols (as opposed to only known constants) which can prove that a loop can execute in parallel.

Runtime Testing

- For some codes, even if it's impossible to determine at compile time whether it's safe to parallelize a loop, that information might be available at runtime.
- Therefore it's common to perform testing at runtime to determine whether a loop should execute sequentially or in parallel.
- In addition to determining the safety of parallel execution, this technique can also be used to determine whether parallel execution will make the loop faster.

Speculative Execution

- Instead of giving up on parallelization of difficult sequential codes, some researchers argue for **speculative execution** at the thread level.
- All superscalar processors speculate on the outcome of branches which is very useful.
- Thread-level speculation might be a good idea but the costs of miss-speculation may be too high.

Back to ILLIAC IV

- Recall ILLIAC IV built from 1965-1975 at the University of Illinois
- As mentioned earlier that project developed many code transformations for vectorization and parallelization.
- The theoretical foundation was made by Utpal Banerjee in his master's thesis.
- He is at Intel and has written a series of very readable books on data dependence analysis and parallelization.

Inner vs Outer Loop Parallelization

- In the course optimizing compilers in LP1 (every even year) we learn about inner loop parallelization which is used e.g. for automatic SIMD vectorization and software pipelining.
- Here the focus instead is on automatic parallelization for multicores, i.e. outer loop parallelization.
- The foundations for inner and outer loop parallelization are similar, since they both rely on data dependence analysis.

Introduction to Data Dependence Analysis

- There are **data dependences** in the following code:

```
S1: x = a + b;
S2: y = x + 1;
S3: x = b * c;
```

- The value written to `x` in S_1 is read in S_2.
- This is called a **true dependence** and is written $S_1 \delta^t S_2$.
- This notation means that S_1 must execute before S_2 in a transformed program.
- In a true data dependence between two statements both statements access the same memory location, and the first statements writes a value which the other statements reads.

Data Dependences at Different Levels

- Data dependences can be at several different levels, including:
  - Instructions
  - Statements
  - Loop iterations
  - Functions
  - Threads
- Parallelizing compilers usually find parallelism between different loop iterations of a loop.
- If the compiler can determine that there are no dependences between loop iterations then it can either:
  - Produce parallel machine code, or
  - Produce source code with OpenMP `#pragma parallel for` directives.
- If there are dependences, it may still be possible to execute the loop in parallel since perhaps the loop iterations are not totally ordered.

Total vs Partial Order and Loop Iteration Iterations

- Integers are totally ordered since we can determine which of a and b is greater if $a \neq b$.
- Consider a directed acyclic graph. In topological sorting you can process any vertex u if all predecessors of u already have been processed.
- Obviously, we should not execute a loop iteration before its input data has been computed.
- In executing a loop in parallel we perform a topological sort of the loop iterations.
- There is no search at runtime to determine which iterations can be executed, though.
- The topological sorting is the major work in parallelization.
- That is, to find an order of the iterations which maximizes parallelism.

Different Types of Data Dependences

- When we write "loop iteration" below our sentence is also valid for the other levels (instruction, statement, etc).
- Below iteration I_1 always executes before iteration I_2.
- Recall, in a **true dependence**, written $I_1 \delta^t I_2$, I_1 produces a value consumed by I_2.
- In an **anti dependence**, written $I_1 \delta^a I_2$, I_1 reads a memory location later overwritten by I_2.
- In an **output dependence**, written $I_1 \delta^o I_2$, I_1 writes a memory location later overwritten by I_2.
- In an **input dependence**, written $I_1 \delta^i I_2$, both I_1 and I_2 read the same memory location.
- The first three types of dependences create partial orderings among all iterations, which parallelizing compilers exploit by ordering iterations to improve performance.

Dependences in the Example Code

- Let us classify all dependences in the code:

```
S1: x = a + b;
S2: y = x + 1;
S3: x = b * c;
```

- There is a true dependence from S1 to S2 due to `x`.
- There is an anti dependence from S2 to S3 due to `x`.
- There is an output dependence from S1 to S3 due to `x`.

Loop Level Data Dependences

- In the loop

```
for (i = 3; i < 100; i += 1)
        a[i] = a[i-3] + x;
```

- There is a true dependence from iteration i to iteration $i + 3$.
- Iteration $i = 3$ writes to a_3 which is read in iteration $i = 6$.
- A loop level true dependence means one iteration writes to a memory location which a later reads.
- Typically this means one iteration I writes to an array element a_x which a **later** iteration reads.

Perfect Loop Nests

- A **perfect loop nest L** is a nest of m nested **for** loops $L_1, L_2, ...L_m$ such that the body of $L_i, i < m$, consists of L_{i+1} and the body of L_m consists of a sequence of assignment statements.
- For $1 < r \leq m$ p_r and q_r are linear functions of $I_1, ..., I_{r-1}$.

for $(I_1 = p_1;\ I_1 <= q_1;\ I_1 + = 1)$ {
    for $(I_2 = p_2;\ I_2 <= q_2;\ I_2 + = 1)$ {
        $\vdots$
        for $(I_m = p_m;\ I_m <= q_m;\ I_m + = 1)$ {
            $h(I_1, I_2, ..., I_m)$;
        }
    }
}

Example Perfect Loop Nest

- All assignments, **except** to the loop index variables are in the innermost loop.
- There may be any number of assignment statements in the innermost loop.

```
for (i = 0; i < 100; i += 1) {
        for (j = 3 + i; j <  2 * i + 10; j += 1) {
                for (k = i - j; k < j - i; k += 1) {
                        a[i][j][k] += b[k][j][i];
                }
        }
}
```

Loop Bounds

- The lower bound for l_1 is $p_{10} \leq l_1$.
- The lower bound for l_2 is

$$\begin{aligned} l_2 &\geq p_{20} + p_{21} l_1 \\ p_{20} &\leq l_2 - p_{21} l_1 \\ p_{20} &\leq -p_{21} l_1 + l_2 \end{aligned} \tag{1}$$

- The lower bound for l_3 is

$$\begin{aligned} l_3 &\geq p_{30} + p_{31} l_1 + p_{32} l_2 \\ p_{30} &\leq l_3 - p_{31} l_1 - p_{32} l_2 \\ p_{30} &\leq -p_{31} l_1 - p_{32} l_2 + l_3 \end{aligned} \tag{2}$$

and so forth. We represent this on matrix form as $\mathbf{p_0} \leq \mathbf{lP}$, or... see next slide.

Loop Bounds on Matrix Form

- $\mathbf{P} = \begin{pmatrix} 1 & -p_{21} & -p_{31} & \dots & -p_{m1} \\ 0 & 1 & -p_{32} & \dots & -p_{m2} \\ 0 & 0 & 1 & \dots & -p_{m3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}$ and $\mathbf{p}_0 = (p_{10}, p_{20}, \dots, p_{m0})$.
- Similarly, the upper bounds are represented as $\mathbf{IQ} \leq \mathbf{q_0}$.
- The loop bounds, thus, are represented by the system:

$$\left.\begin{array}{ccccc} \mathbf{p_0} & \leq & \mathbf{IP} & & \\ & & \mathbf{IQ} & \leq & \mathbf{q_0} \end{array}\right\}$$

Example Non-Perfect Loop Nest

- The assignment to c_{ij} before the innermost loop makes it a non-perfect loop nest.
- Sometimes non-perfect loop nest can be split up, or **distributed** into perfect loop nests.
- See next slide.

```
for (i = 0; i < 100; i += 1) {
        for (j = 0; j < 100; j += 1) {
                c[i][j] = 0;
                for (k = 0; k < 100; k += 1) {
                        c[i][j] += a[i][k] * b[k][j];
                }
        }
}
```

Loop Distribution

- Result of loop distribution.

```
for (i = 0; i < 100; i += 1)
        for (j = 0; j < 100; j += 1)
                c[i][j] = 0;
for (i = 0; i < 100; i += 1)
        for (j = 0; j < 100; j += 1)
                for (k = 0; k < 100; k += 1)
                        c[i][j] += a[i][k] * b[k][j];
```

Some Terminology

- The index vector $\mathbf{I} = (I_1, I_2, ..., I_m)$ is the vector of index variables.
- The index values of **L** are the values of $(I_1, I_2, ..., I_m)$.
- The index space of **L** is the subspace of Z^m consisting of all the index values.
- An **affine array reference** is an array reference in which all subscripts are linear functions of the loop index variables.

Symbolic Analysis

- Data dependence analysis is normally restricted to affine array references.
- In practice, however, subscripts often contain **symbolic constants** as shown below which is test `s171` in the C version of the Argonne Test Suite for Vectorising Compilers.
- There is no dependence between the iterations in this test.

```
for (i=0; i<n; i++)
    a[i*n] = a[i*n] + b[i];
```

Problematic Non-Affine Index Functions

- In the loop

```
scanf("%d", &x);

for (i = 3; i < 100; i += 1) {
S1:     a[i]   = a[x] + 1;
S2:     b[i]   = b[c[i-1]] + 2;
S3:     d[i]   = d[i * i] + 3;
}
```

- Some compilers try to do runtime dependence testing to take care of S_1 but it may cause too much overhead if many variables must be checked.
- While S_3 is not difficult, almost all parallelizing compilers focus on index expressions which are linear functions of the loop variables.

Representing Array References

- Let X be an n-dimensional array. Then an affine reference has the form:
- $X[a_{11}i_1 + a_{21}i_2 ... a_{m1}i_m + a_{01}]...[a_{1n}i_1 + a_{2n}i_2 ... a_{mn}i_m + a_{0n}]$
- This is conveniently represented as a matrix and a vector $X[\mathbf{IA} + \mathbf{a_0}]$, where
- $\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$ and $\mathbf{a}_0 = (a_{10}, a_{20}, ..., a_{n0})$.
- We will refer to $\mathbf{A}$ and $\mathbf{a_0}$ as the **coefficient matrix** and the **constant term**, respectively.

The Data Dependence Equation

- For two references $X[\mathbf{IA} + \mathbf{a_0}]$ and $X[\mathbf{IB} + \mathbf{b_0}]$ to refer to the same array element there must be two index values, $\mathbf{i}$ and $\mathbf{j}$ such that $\mathbf{iA} + \mathbf{a_0} = \mathbf{jB} + \mathbf{b_0}$ which we can write as $\mathbf{iA} - \mathbf{jB} = \mathbf{b_0} - \mathbf{a_0}$.
- This system of Diophantine equations has n (the dimension of the array X) scalar equations and $2m$ variables, where m is the nesting depth of the loop.
- It can also be written in the following form:

$$(\mathbf{i}; \mathbf{j}) \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{b_0} - \mathbf{a_0}. \tag{3}$$

- We solve the system of linear Diophantine equations in (3) using a method presented shortly.

An Example

```
for (i = 0; i < 100; i += 1)
        for (j = 2*i + 4; j < i + 40; j += 1)
                a[2i-3j-1][2i+j-3] = f(a[-3i+4j+1][-i+2j+7]);
```

- The above loop nest has the following two array reference representations:
  $\mathbf{A} = \begin{pmatrix} 2 & 2 \\ -3 & 1 \end{pmatrix}$ and $\mathbf{a}_0 = (-1, -3)$.
  $\mathbf{B} = \begin{pmatrix} -3 & -1 \\ 4 & 2 \end{pmatrix}$ and $\mathbf{b}_0 = (1, 7)$.

Dependence Distances

- Let $\prec_\ell$ be a relation in $\mathbf{Z}^m$ such that $\mathbf{i} \prec \mathbf{j}$ if $i_1 = j_1$, $i_2 = j_2$, ..., $i_{l-1} = j_{l-1}$, and $i_l < j_l$.
- For example: $(1, 3, 4) \prec_3 (1, 3, 9)$.
- The lexicographic order $\prec$ in $\mathbf{Z}^m$ is the union of all the relations $\prec_\ell$: $\mathbf{i} \prec \mathbf{j}$ iff $\mathbf{i} \prec_\ell \mathbf{j}$ for some ℓ in $1 \leq \ell \leq m$.
- The sequential execution of the iterations of a loop nest follows the lexicographic order.
- Assume that $(\mathbf{i}; \mathbf{j})$ is a solution to (3), and that $\mathbf{i} \prec \mathbf{j}$. Then $\mathbf{d} = \mathbf{j} - \mathbf{i}$ is the **dependence distance** of the dependence.

Uniform Dependence Distance

- If a dependence distance **d** is a constant vector then the dependence is said to be uniform.
- The dependence distance $\mathbf{d} = (1, 2)$ is uniform, while the dependence distance $\mathbf{d} = (1, t_2)$ is nonuniform.
- Uniform distance vectors are *very* desirable since loops with only uniform distance vectors can be optimized with unimodular transformations, described below.
- For this, the set of all distance vectors $\mathbf{d_i}$ of a loop nest **L** are arranged in a matrix with *n* rows and *m* columns where *n* is the number of dependences in **L** and *m* is the number of index variables in **L**.
- Note that a zero distance between references within the same statement does not cause an loop-level dependence, e.g. `a[i] = a[i] + x` but still an instruction-level dependence.

Loop Independent and Loop Carried Dependences

- A loop independent dependence is a dependence such that $\mathbf{d} = \mathbf{j} - \mathbf{i} = (0, ..., 0)$.
- A loop independent dependence does not prevent concurrent execution of different iterations of a loop. Rather, it constrains the scheduling of instructions in the loop body.
- A loop carried dependence is a dependence which is not loop independent, or, in other words, the dependence is between two different iterations of a loop nest.
- A dependence has level ℓ if in $\mathbf{d} = \mathbf{j} - \mathbf{i}$, $\mathbf{d}_1 = 0, \mathbf{d}_2 = 0, ..., \mathbf{d}_{l-1} = 0$, and $\mathbf{d}_l > 0$.
- Only a loop carried dependence has a level, and it is only the loop at that level which needs to be executed sequentially.

The GCD Test

- The GCD test was invented at Texas Instruments and first described 1973.
- Consider the loop

```
for (i = lb; i <= ub; ++i)
        x[ a1 * i + c1] = x[a2 * i + c2] + y;
```

- To prove independence, we must show that the Diophantine equation

$$a_1 i_1 - a_2 i_2 = c_2 - c_1 \tag{4}$$

has no solutions.

- We compute the gcd of a_1 and a_2 and check whether it divides $c_2 - c_1$, and if it does not, there is no solution and we have proved independence, otherwise we must use another test.

Weaknesses of The GCD Test

- There are two weaknesses of the GCD test:
  1. It does not exploit knowledge about the loop bounds.
  2. Most often the gcd is one.
- The first weakness means the GCD Test might be unable to prove independence despite the solution to (4) actually lies outside the index space of the loop.
- The second weakness means independence usually cannot be proved.

GCD Test for Nested Loops and Multdimensional Arrays

- The GCD Test can be extended to cover nested loops and multidimensional arrays.
- The solution is then a vector and it usually contains unknowns.
- The Fourier-Motzkin Test described shortly takes the solution vector from this GCD Test and checks whether the solution lies within the loop bounds.
- Next we will look at unimodular matrices and Fourier-Motzkin elimination used by the Fourier-Motzkin Test.

Unimodular Matrices

- An integer square matrix $\mathbf{A}$ is unimodular if its determinant $det(\mathbf{A}) = \pm 1$.
- If $\mathbf{A}$ and $\mathbf{B}$ are unimodular, then $\mathbf{A}^{-1}$ exists and is itself unimodular, and $\mathbf{A} \times \mathbf{B}$ is unimodular.
- $\mathcal{I}$ is the $m \times m$ identity matrix.

Elementary Row Operations

- The operations
  - *reversal*: multiply a row by -1,
  - *interchange*: interchange two rows, and
  - *skewing*: add an integer multiple of one row to another row,

  are called the elementary row operations.
- With each elementary row operation, there is a corresponding *elementary matrix*.

Performing Elementary Row Operations

- To perform an elementary row operation on a matrix $\mathbf{A}$, we can premultiply it with the corresponding elementary matrix.
- Assume we wish to interchange rows 1 and 3 in a 3×3 matrix $\mathbf{A}$. The resulting matrix is formed by

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \times \mathbf{A}.$$

- The elementary matrices are all unimodular.

3×3 Reversal Matrices

- $$\begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

  and
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}.$$

3×3 Interchange Matrices

- $$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix},$$

  and
- $$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

3×3 Upper Skewing Matrices

- $$\begin{pmatrix} 1 & z & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

- $$\begin{pmatrix} 1 & 0 & z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

  and

- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & z \\ 0 & 0 & 1 \end{pmatrix}.$$

3×3 Lower Skewing Matrices

- $$\begin{pmatrix} 1 & 0 & 0 \\ z & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ z & 0 & 1 \end{pmatrix},$$

  and
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & z & 1 \end{pmatrix}.$$

Echelon Matrices

- Let l_i denote the column number of the first nonzero element of matrix row i.
- A given $m \times n$ matrix $\mathbf{A}$, is an *echelon matrix* if the following are satisfied for some integer ρ in $0 \leq \rho \leq m$:
    - rows 1 through ρ are nonzero rows,
    - rows $\rho + 1$ through m are zero rows,
    - for $1 \leq i \leq \rho$, each element in column l_i below row i is zero, and
    - $l_1 < l_2 < ... < l_\rho$.
- The following are examples of echelon matrices:

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 0 & 0 & 4 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \\ 0 & 0 & 0 \end{pmatrix}$$

Echelon Reduction

- Given an $m \times n$ matrix $\mathbf{A}$, Echelon reduction finds two matrices $\mathbf{U}$ and $\mathbf{S}$ such that $\mathbf{U} \times \mathbf{A} = \mathbf{S}$, where $\mathbf{U}$ is unimodular and $\mathbf{S}$ is echelon.
- $\mathbf{U}$ remains unimodular since we only apply elementary row operations.

```
function echelon_reduce(A)
        U ← I_m
        S ← A
        i_0 ← 0
        for (j ← 1; j ≤ n; j ← j + 1) {
            if (there is a nonzero s_ij with i_0 < i ≤ m) {
                i_0 ← i_0 + 1
                i = m
                while (i ≥ i_0 + 1) {
                    while (s_ij ≠ 0) {
                        σ ← sign(s_(i−1)j × s_ij)
                        z ← ⌊|s_(i−1)j| / |s_ij|⌋
                        subtract σz(row i) from (row i − 1) in (U; S)
                        interchange rows i and i − 1 in (U; S)
                    }
                    i ← i − 1
                }
            }
        }
    return U and S
end
```

Example Echelon Reduction

- We will now show how one can echelon reduce the following matrix:

$$\mathbf{A} = \begin{pmatrix} 2 & 2 \\ -3 & 1 \\ 3 & 1 \\ -4 & -2 \end{pmatrix}.$$

- We start with with $\mathbf{U} = \mathbf{I_4}$ and $\mathbf{S} = \mathbf{A}$ which we write as:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 1 & 0 & 3 & 1 \\ 0 & 0 & 0 & 1 & -4 & -2 \end{array}\right).$$

- Then we will eliminate the nonzero elements in $\mathbf{S}$ starting with $s_{41}, s_{31}, s_{21}, s_{42}$ and so on.

Example Echelon Reduction

- $\mathbf{j = 1, i_0 = 1, i = 4}$. We always wish to eliminate s_{ij}, which currently means s_{41}.
- $\sigma \leftarrow -1$ and $z \leftarrow 0$. Nothing is subtracted from row 3.
- Then rows 3 and 4 are interchanged in $(\mathbf{U}; \mathbf{S})$, resulting in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|rr} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 0 & 1 & -4 & -2 \\ 0 & 0 & 1 & 0 & 3 & 1 \end{array} \right).$$

Example Echelon Reduction

- We continue the inner while loop and find that $\sigma \leftarrow -1$ and $z \leftarrow 1$. Then $-1\times$ row 4 is subtracted from row 3, resulting in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 1 & 0 & 3 & 1 \end{array}\right).$$

- Then rows 3 and 4 are interchanged, resulting in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 1 & 0 & 3 & 1 \\ 0 & 0 & 1 & 1 & -1 & -1 \end{array}\right).$$

Example Echelon Reduction

- s_{41} is still zero, and the inner while loop is continued and $\sigma \leftarrow -1$ and $z \leftarrow 3$. Then $-3\times$ row 4 is subtracted from row 3:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 4 & 3 & 0 & -2 \\ 0 & 0 & 1 & 1 & -1 & -1 \end{array}\right).$$

- Then rows 3 and 4 are interchanged, resulting in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

- Now the first $_{ij}$ has become zero and i is decremented.

Example Echelon Reduction

- $\mathbf{j = 1, i_0 = 1, i = 3}$. We now wish to eliminate s_{31}. $\sigma \leftarrow +1$ and $z \leftarrow 3$. Then $3\times$ row 3 is subtracted from row 2:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{rrrr|rr} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & -3 & -3 & 0 & 4 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

- Then rows 2 and 3 are interchanged, resulting in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{rrrr|rr} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 1 & -3 & -3 & 0 & 4 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

Example Echelon Reduction

- $\mathbf{j = 1, i_0 = 1, i = 2}$. We now wish to eliminate s_{21}. $\sigma \leftarrow -1$ and $z \leftarrow 2$. Then $-2\times$ row 2 is subtracted from row 1:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 1 & -3 & -3 & 0 & 4 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

- Interchanging rows 2 and 1 results in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & -3 & -3 & 0 & 4 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

Example Echelon Reduction

- $\mathbf{j = 2, i_0 = 2, i = 4}$. We now wish to eliminate s_{42}. $\sigma \leftarrow -1$ and $z \leftarrow 2$. $-2\times$ row 4 is subtracted from row 3:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & 5 & 3 & 0 & 0 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

- Interchanging rows 4 and 3 results in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 0 & 4 & 3 & 0 & -2 \\ 0 & 1 & 5 & 3 & 0 & 0 \end{array}\right).$$

Example Echelon Reduction

- $\mathbf{j = 2, i_0 = 2, i = 3}$. We now wish to eliminate s_{32}. $\sigma \leftarrow 0$ and $z \leftarrow 0$. Nothing is subtracted from row 2 but rows 3 and 2 are interchanged:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 4 & 3 & 0 & -2 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & 5 & 3 & 0 & 0 \end{array}\right).$$

At this point $\mathbf{S}$ is an echelon matrix and the algorithm stops (the outer while loop since $i = i_0$). As will turn out to be convenient later, we prefer positive values of s_{11} and therefore multiply with -1 finally resulting in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 0 & 0 & -1 & -1 & 1 & 1 \\ 0 & 0 & 4 & 3 & 0 & -2 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & 5 & 3 & 0 & 0 \end{array}\right).$$

GCD of multiple integers

- Let $a_1, a_2, ..., a_m$ denote a list of integers, not all zero,
- $\mathbf{U}$ an $m \times m$ unimodular matrix,
- $\mathbf{S} = (s_{11}, 0, ...0)^T$ an $m \times 1$ echelon matrix, such that $\mathbf{UA} = \mathbf{S}$ where $\mathbf{A}$ is the $m \times 1$ matrix $(a_1, a_2, ..., a_m)^T$,
- then $\gcd(a_1, a_2, ..., a_m) = |s_{11}|$.

Linear Diophantine Equations

- To perform data dependence analysis for multidimensional arrays we need to consider a system of n linear diophantine equations in m variables.
- m is twice the loop nesting and n the number of dimensions in an array.

$$\mathbf{xA} = \mathbf{c} \tag{5}$$

  Here $\mathbf{x}$ is an $1 \times m$ integer matrix, $\mathbf{A}$ is an $m \times n$ integer matrix, and $\mathbf{c}$ is an $1 \times n$ integer matrix.
- (5) is easy to solve if $\mathbf{A}$ is an echelon matrix.
- With echelon reduction we find $\mathbf{U}$ and $\mathbf{S}$ such that $\mathbf{UA} = \mathbf{S}$.
- We will check if there is an integer solution to $\mathbf{tS} = \mathbf{c}$ instead.

Linear Diophantine Equations

Theorem

- *Let* $\mathbf{A}$ *be a given* $m \times n$ *integer matrix and* $\mathbf{c}$ *a given integer* n *vector.*
- *Let* $\mathbf{U}$ *denote an* $m \times m$ *integer matrix an* $\mathbf{S}$ *an* $m \times n$ *integer echelon matrix, such that* $\mathbf{UA} = \mathbf{S}$*.*

The system of equations

$$\mathbf{xA} = \mathbf{c} \tag{6}$$

has a solution iff there exists an integer m-vector $\mathbf{t}$ *such that* $\mathbf{tS} = \mathbf{c}$*. When a solution exists, the set of all solutions is given by the formula*

$$\mathbf{x} = \mathbf{tU} \tag{7}$$

where $\mathbf{t}$ *is the integer vector which satisfies* $\mathbf{tS} = \mathbf{c}$*.*

Linear Diophantine Equations

Proof.

- An integer m-vector $\mathbf{x} = \mathbf{tU}$ will be a solution to 7 iff

$$\mathbf{c} = \mathbf{xA} = \mathbf{tUA} = \mathbf{tS} \tag{8}$$

- If there is no such integer vector $\mathbf{t}$ such that $\mathbf{tS} = \mathbf{c}$, then there is no integer solution to $\mathbf{xA} = \mathbf{c}$ either.
- If there is such a $\mathbf{t}$, then all solutions have the form $\mathbf{x} = \mathbf{tU}$, where $\mathbf{t}$ is integral and $\mathbf{tS} = \mathbf{c}$.

$\square$

Linear Diophantine Equations

- To illustrate how equations of the form $\mathbf{xA} = \mathbf{c}$ can be solved using the techniques introduced above, let us solve

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 \end{pmatrix} \begin{pmatrix} 2 & 2 \\ -3 & 1 \\ 3 & 1 \\ -4 & -2 \end{pmatrix} = \begin{pmatrix} 2 & 4 \end{pmatrix} \tag{9}$$

- Firstly we use echelon reduction to find the matrices $\mathbf{U}$ and $\mathbf{S}$.
- Then we formulate the equation $\mathbf{tS} = \mathbf{c}$:

$$\begin{pmatrix} t_1 & t_2 & t_3 & t_4 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & -2 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 4 \end{pmatrix} \tag{10}$$

  It is trivially solved and we find that $\mathbf{t} = (2, -1, t_3, t_4)$, where t_3 and t_4 are arbitrary integers.

Linear Diophantine Equations

- We then find $\mathbf{x}$:

$$\mathbf{x} = \mathbf{tU} = \begin{pmatrix} 2 & -1 & t_3 & t_4 \end{pmatrix} \begin{pmatrix} 0 & 0 & -1 & -1 \\ 0 & 0 & 4 & 3 \\ 1 & 0 & 2 & 2 \\ 0 & 1 & 5 & 3 \end{pmatrix} = \tag{11}$$

$$(t_3, t_4, 2t_3 + 5t_4 - 7, 2t_3 + 3t_4 - 5) \tag{12}$$

Fourier-Motzkin Elimination

- Suppose we find, during data dependence analysis, an integer vector $\mathbf{x}$ which is a solution to $\mathbf{xA} = \mathbf{c}$.
- Then what we can conclude is that there exist index variables such that the two array references being tested can reference the same memory location.
- If no solution can be found then we know there is no dependence. If there is a solution, then there may be a dependence.
- If the solution $\mathbf{x}$ represents index variables which are outside the loop bounds, then $\mathbf{x}$ does not prove that a data dependence exists. So, we need also solve a system of linear inequalities when the solution $\mathbf{x}$ exists.
- An additional constraint is, of course, that the solution is integer. Unfortunately, the problem of solving a linear integer inequality is NP-complete.

Fourier-Motzkin Elimination

- In 1827 Fourier published a method for solving linear inequalities in the real case. This method is known as Fourier-Motzkin elimination and is used in compilers as an approximation.
- If Fourier-Motkin elimination finds that there is no real solution, then there certainly is no integer either. But if there is a real solution, there may or may not be an integer solution.
- Fourier-Motzkin elimination is regarded as a time-consuming algorithm and to apply it so perhaps thousands of data dependence tests may make the compiler too slow. Therefore, it is used as a backup tests when other faster tests fail to prove independence.

Fourier-Motzkin Elimination

- An interesting question is how frequently Fourier-Motzkin elimination finds a real solution when there is no integer solution. Some special cases can be exploited.
- For instance, if a variable x_i must satisfy $2.2 \leq x_i \leq 2.8$ then there is no integer solution.
  Otherwise, if we find eg that $2.2 \leq x_i \leq 4.8$ then we may try the two cases of setting $x_i = 3$ and $x_i = 4$, and see if there still is a real solution.
- It is easiest to understand Fourier-Motzkin elimination if we first look at an example.

Fourier-Motzkin Elimination

- Assume we wish to solve the following system of linear inequalities.

$$\begin{array}{rcrcr} 2x_1 & - & 11x_2 & \leq & 3 \\ -3x_1 & + & 2x_2 & \leq & -5 \\ x_1 & + & 3x_2 & \leq & 4 \\ -2x_1 & & & \leq & -3 \end{array} \tag{13}$$

- We will first eliminate x_2 from the system, and then check whether the remaining inequalities can be satisfied. To eliminate x_2, we start out with sorting the rows with respect to the coefficients of x_2:

$$\begin{array}{rcrcr} -3x_1 & + & 2x_2 & \leq & -5 \\ x_1 & + & 3x_2 & \leq & 4 \\ 2x_1 & - & 11x_2 & \leq & 3 \\ -2x_1 & & & \leq & -3 \end{array} \tag{14}$$

Fourier-Motzkin Elimination

- First we want to have rows with positive coefficients of x_2, then negative, and lastly zero coefficients.
- Next we divide each row by its coefficient (if it is nonzero) of x_2:

$$\begin{array}{rcrcr} \frac{-3}{2}x_1 & + & x_2 & \leq & \frac{-5}{2} \\ \frac{1}{3}x_1 & + & x_2 & \leq & \frac{4}{3} \\ \frac{2}{11}x_1 & - & x_2 & \geq & \frac{3}{11} \end{array} \tag{15}$$

Of course, the $\leq$ becomes $\geq$ when dividing with a negative coefficient. We can now rearrange the system to isolate x_2:

$$\begin{array}{rcrcrcrcr} & & & & x_2 & \leq & \frac{3}{2}x_1 & - & \frac{5}{2} \\ & & & & x_2 & \leq & -\frac{1}{3}x_1 & + & \frac{4}{3} \\ \frac{2}{11}x_1 & - & \frac{3}{11} & \leq & x_2 & & & & \end{array} \tag{16}$$

- At this point, we make a record of the minimum and maximum values that x_2 can have, expressed as functions of x_1. We have:

$$b_2(x_1) \leq x_2 \leq B_2(x_1) \tag{17}$$

where

$$\begin{aligned} b_2(x_1) &= \frac{2}{11}x_1 \\ B_2(x_1) &= \min\left(\frac{3}{2}x_1 - \frac{5}{2}, -\frac{1}{3}x_1 + \frac{4}{3}\right) \end{aligned} \tag{18}$$

Fourier-Motzkin Elimination

- To eliminate x_2 from the system, we simply combine the inequalities which had positive coefficients of x_2 with those which had negative coefficients (ie, one with positive coefficient is combined with one with negative coefficient):

$$\begin{array}{rcrcrcr} \frac{2}{11}x_1 & - & \frac{3}{11} & \leq & \frac{3}{2}x_1 & - & \frac{5}{2} \\ \frac{2}{11}x_1 & - & \frac{3}{11} & \leq & -\frac{1}{3}x_1 & + & \frac{4}{3} \end{array} \tag{19}$$

- These are simplified and the inequality with the zero coefficient of x_2 is brought back:

$$\begin{array}{rcr} -\frac{29}{22}x_1 & \leq & -\frac{49}{22} \\ -\frac{17}{33}x_1 & \leq & \frac{53}{33} \\ -2x_1 & \leq & -3 \end{array} \tag{20}$$

Fourier-Motzkin Elimination

- We can now repeat parts of the procedure above:

$$\begin{aligned} x_1 &\leq \frac{53}{17} \\ x_1 &\geq \frac{49}{29} \\ x_1 &\geq \frac{3}{2} \end{aligned} \tag{21}$$

- We find that

$$\begin{aligned} b_1() &= \max(49/29, 3/2) = 49/29 \\ B_1() &= 53/17 \end{aligned} \tag{22}$$

The solution to the system is $\frac{49}{29} \leq x_1 \leq \frac{53}{17}$ and $b_2(x_1) \leq B_2(x_1)$ for each value of x_1.

Fourier-Motzkin Elimination

procedure *fourier_motzkin_elimination* (x, A, c)
    $r \leftarrow m, \quad s \leftarrow n, \quad \mathbf{T} \leftarrow \mathbf{A}, \quad \mathbf{q} \leftarrow \mathbf{c}$
    while (1) {
        $n_1 \leftarrow$ number of inqualities with positive t_{rj}
        $n_2 \leftarrow n_1 +$ number of inqualities with negative t_{rj}
        Sort the inequalities so that the n_1 with $t_{rj} > 0$ come first,
            then the $n_2 - n_1$ with $t_{rj} < 0$ come next,
            and the ones with $t_{rj} = 0$ come last.
        for $(i = 1;\ i \leq r - 1;\ i \leftarrow i + 1)$
            for $(j = 1;\ i \leq n_2;\ j \leftarrow j + 1)$
                $t_{ij} \leftarrow t_{ij} / t_{rj}$
        for $(j = 1;\ i \leq n_2;\ j \leftarrow j + 1)$
            $q_j \leftarrow q_j / t_{rj}$
        if $(n_2 > n_1)$
            $b_r(x_1, x_2, \ldots, x_{r-1}) = \max_{n_1+1 \leq j \leq n_2}(-\sum_{i=1}^{r-1} t_{ij}x_i + q_i)$
        else
            $b_r \leftarrow -\infty$
        if $(n_1 > 0)$
            $j_r(x_1, x_2, \ldots, x_{r-1}) = \min_{n_1+1 \leq j \leq n_2}(-\sum_{i=1}^{r-1} t_{ij}x_i + q_i)$
        else
            $B_r \leftarrow \infty$
        if $(r = 1)$
            return *make_solution()*

Fourier-Motzkin Elimination

/* We will now eliminate x_r. */
$s' \leftarrow s - n_2 + n_1(n_2 - n_1)$
if ($s' = 0$) {
/* We have not discovered any inconsistency and */
/* we have no more inequalities to check. */
/* The system has a solution. */
The solution set consists of all real vectors $(x_1, x_2, \ldots, x_m)$,
where $x_{r-1}, x_{r-2}, \ldots, x_1$ are chosen arbitrarily, and
$x_m, x_{m-1}, \ldots, x_r$ must satisfy
$b_i(x_1, x_2, \ldots, x_{i-1}) \leq x_i \leq B_i(x_1, x_2, \ldots, x_{i-1})$ for $r \leq i \leq m$.
return solution set.
}
/* There are now s' inequalities in $r - 1$ variables. */
The new system of inequalities is made of two parts:
$\sum_i^{r-1}(t_{ik} - t_{il})x_i \leq q_k - q_j$ for $1 \leq k \leq n_1, n_1 + 1 \leq j \leq n_2$
$\sum_i^{r-1} t_{ij}x_i \leq q_j$ for $n_2 + 1 \leq j \leq s$
and becomes by setting $r = r \leftarrow 1$ and $s \leftarrow s'$:
$\sum_i^r t_{ij}x_i \leq q_j$ for $1 \leq j \leq s$
} **end**

function *make_solution* ()
/* We have come to the last variable x_1. */
if ($b_1 > B_1$ **or** (there is a $q_j < 0$ for $n_2 + 1 \leq j \leq s$))
return there is no solution
The solution set consists of all real vectors $(x_1, x_2, \ldots, x_m)$,
such that $b_i(x_1, x_2, \ldots, x_m) \leq x_i \leq B_i(x_1, x_2, \ldots, x_m)$ for $1 \leq i \leq m$.
return solution set.
end

Summary

- In the case of a loop nest of height m and an n-dimensional array, we use the matrix representation of the references $\mathbf{iA} + \mathbf{a_0} = \mathbf{jB} + \mathbf{b_0}$, or equivalently:

$$(\mathbf{i}; \mathbf{j}) \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{b_0} - \mathbf{a_0}, \tag{23}$$

  where the $\mathbf{A}$ and $\mathbf{B}$ have m rows and n columns.
- We find a $2m \times 2m$ unimodular matrix $\mathbf{U}$ and a $2m \times n$ echelon matrix $\mathbf{S}$ such that

$$\mathbf{U} \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{S}. \tag{24}$$

- If there is a $2m$ vector $\mathbf{t}$ which satisfies $\mathbf{tS} = \mathbf{b_0} - \mathbf{a_0}$ then the GCD test cannot exclude dependence, and if so...
- ..., the computed $\mathbf{t}$ will be input to the Fourier-Motzkin Test.

The Fourier-Motzkin Test

- If the GCD Test found a solution vector $\mathbf{t}$ to $\mathbf{tS} = \mathbf{c}$, these solutions will be tested to see if they are within the loop bounds.
- Recall we wrote

$$\mathbf{x} = (\mathbf{i}; \mathbf{j}) \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{b_0} - \mathbf{a_0}. \tag{25}$$

- We find $\mathbf{x}$ from:

$$\mathbf{x} = (\mathbf{i}; \mathbf{j}) = \mathbf{tU} \tag{26}$$

- With $\mathbf{U_1}$ being the left half of $\mathbf{U}$ and $\mathbf{U_2}$ the right half we have:

$$\mathbf{i} = \mathbf{tU_1} \tag{27}$$

$$\mathbf{j} = \mathbf{tU_2} \tag{28}$$

- These should be inserted to loop bounds constraints.

The Fourier Motzkin Test

- Recall the original loop bounds are:

$$\left.\begin{array}{ccccc} \mathbf{p_0} & \leq & \mathbf{IP} & & \\ & & \mathbf{IQ} & \leq & \mathbf{q_0} \end{array}\right\}$$

- The solution vector $\mathbf{t}$ must satisfy:

$$\left.\begin{array}{ccccc} \mathbf{p_0} & \leq & \mathbf{tU_1P} & & \\ & & \mathbf{tU_1Q} & \leq & \mathbf{q_0} \\ \mathbf{p_0} & \leq & \mathbf{tU_2P} & & \\ & & \mathbf{tU_2Q} & \leq & \mathbf{q_0} \end{array}\right\} \tag{29}$$

- If there is no integer solution to this system, there is no dependence.
- Recall, however, the system is solved with real or rational numbers so the Fourier-Motzkin Test may fail to exclude independence.

After Data Dependence Analysis

- When we have performed data dependence analysis of all pairs of references to the same arrays, we have a **dependence matrix**, denoted **D**.
- Some rows will be due to some array and other rows due to some other arrays.
- It's the dependence matrix that determines which transformations we can do.
- As mentioned, in the optimizing compilers course inner loop transformations are studied for SIMD vectorization and software pipelining.
- We will look at outer loop parallelization.

Unimodular Transformations

- A **unimodular transformation** is a loop transformation completely expressed as a unimodular matrix $\mathbf{U}$.
- A loop nest $\mathbf{L}$ is changed to a new loop nest $\mathbf{L_U}$ with loop index variables:

$$\mathbf{K} = \mathbf{IU}$$
$$\mathbf{I} = \mathbf{KU}^{-1}$$

- The same iterations are executed but in a different order.
- A new iteration order might make parallel execution possible.
- Before generating code for the new loop, the loop bounds for $\mathbf{K}$ must be computed from the original bounds:

$$\left.\begin{array}{ccccc} \mathbf{p_0} & \leq & \mathbf{IP} & & \\ & & \mathbf{IQ} & \leq & \mathbf{q_0} \end{array}\right\}$$

Computing the New Index Variables

- With

$$\left.\begin{array}{rcccl} \mathbf{p_0} & \leq & \mathbf{IP} & & \\ & & \mathbf{IQ} & \leq & \mathbf{q_0} \end{array}\right\} \tag{30}$$

$$\mathbf{I} = \mathbf{KU}^{-1} \tag{31}$$

We use Fourier-Motzkin elimination to find the loop bounds from

$$\left.\begin{array}{rcccl} \mathbf{p_0} & \leq & \mathbf{KU}^{-1}\mathbf{P} & & \\ & & \mathbf{KU}^{-1}\mathbf{Q} & \leq & \mathbf{q_0} \end{array}\right\} \tag{32}$$

- The bounds are found starting with k_1, k_2 etc.
- This is the reason why we want to have an invertible transformation matrix.

New Array References

- All array references are rewritten to use the new index variables.
- Conceptually we could calculate, at the beginning of each loop iteration,
$$\mathbf{I} = \mathbf{K}\mathbf{U}^{-1}$$
and then use this vector $\mathbf{I}$ in the original references, on the form:
$$x[\mathbf{I}\mathbf{A} + \mathbf{a_0}]$$
- We don't do that of course and instead replace each reference with
$$x[\mathbf{K}\mathbf{U}^{-1}\mathbf{A} + \mathbf{a_0}]$$
- Here $\mathbf{K}\mathbf{U}^{-1}\mathbf{A} + \mathbf{a_0}$ can be calculated at compile-time.

The Distance Matrix

- The set of all vectors of dependence distances is represented by the **distance matrix D**.
- We are free to swap the rows of **D** since it really is a set of dependences.
- Unimodular transformations require that all dependences are uniform, i.e. with known constants.
- Consider a uniform dependence vector $\mathbf{d} = \mathbf{j} - \mathbf{i}$.
- With the **K** index variables we have $\mathbf{d_U} = \mathbf{jU} - \mathbf{iU} = \mathbf{dU}$.
- Therefore, given a dependence matrix **D** and a unimodular transformation **U**, the dependences in the new loop $\mathbf{L_U}$ become:

$$\mathbf{D_U} = \mathbf{DU}$$

Valid Distance Matrices

- The sign, **lexicographically**, of a vector is the sign of the first nonzero element.
- A distance vector can never be lexicographically negative since it would mean that some iteration would depend on a future iteration.
- Therefore no row in the new distance matrix $\mathbf{D_U} = \mathbf{DU}$ may be lexicographically negative.
- If we would discover a lexicographically negative row in $\mathbf{D_U}$, that loop transformation is invalid, such as the second row of the following $\mathbf{D_U}$:

$$D_U = \begin{pmatrix} 1 & 2 \\ -1 & 1 \end{pmatrix}$$

Outer Loop Parallelization

- By **outer loops** is meant all loops starting with the outermost loop.
- While we always can find a unimodular matrix through which we can parallelize the inner loops, this is not the case for outer loops.
- To parallelize the inner loops, we need to assure that all loop carried dependences are carried at the outermost loop.
- In other words, the leftmost column of the distance matrix $\mathbf{D_U}$ simply should consist only of positive numbers!
- For outer loop parallelization, $\mathbf{D_U}$ instead should have leading zero columns.

Rank of a Matrix

- A column of a matrix is linearly independent if it cannot be expressed as a linear combination of the other columns.
- The rank of a matrix is the number of linearly independent columns.
- For instance, an identity matrix $\mathbf{I_m}$ with m columns has $\text{rank}(\mathbf{I_m}) = m$.
- Any unimodular $m \times m$-matrix $\mathbf{U}$ has $\text{rank}(\mathbf{U}) = m$.
- A matrix with zero columns must have a rank less than the number of columns.
- So, since $\mathbf{D_U} = \mathbf{DU}$, if $\mathbf{D_U}$ should have a rank less than m, it must be $\mathbf{D}$ which contributes with that.

Outer Loop Parallelization Example

- Assume we have the distance matrix $\mathbf{D}$ defined as:

$$\mathbf{D} = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix}$$

- With this distance matrix, only the innermost loop can be executed in parallel.
- We want a $\mathbf{D_U}$ with positive rows and zero columns to the left.
- For example:

$$\mathbf{D_U} = \begin{pmatrix} 0 & ? & ? \\ 0 & ? & ? \\ 0 & ? & ? \end{pmatrix} = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \mathbf{U}$$

- If $rank(\mathbf{D}) = 3$ then such a $\mathbf{U}$ cannot exist.

Steps towards Finding $\mathbf{U}$

- We start with transposing $\mathbf{D}$:

$$\mathbf{D^t} = \begin{pmatrix} 6 & 0 & 1 \\ 4 & 1 & 0 \\ 2 & -1 & 1 \end{pmatrix}$$

- Using the Echelon reduction algorithm, we compute:
  - a unimodular matrix $\mathbf{V}$
  - an echelon matrix $\mathbf{S}$
- Such that $\mathbf{VD^t} = \mathbf{S}$, e.g.

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & -2 \\ 1 & -1 & -1 \end{pmatrix} \mathbf{D^t} = \begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -2 \\ 0 & 0 & 0 \end{pmatrix}$$

More Steps towards Finding **U**

- We have $\mathbf{V}\mathbf{D}^{\mathbf{t}} = \mathbf{S}$:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & -2 \\ 1 & -1 & -1 \end{pmatrix} \begin{pmatrix} 6 & 0 & 1 \\ 4 & 1 & 0 \\ 2 & -1 & 1 \end{pmatrix} = \begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -2 \\ 0 & 0 & 0 \end{pmatrix}$$

- Assume we wish to find $n = 1$ parallel outer loops.
- Then we find an $m \times (n+1)$ matrix **A** such that **DA** has n zero columns and then a column with elements greater than zero.
- This **A** will be used to find **U**.
- How can we find **A**?
- Multiplying the last row of **V** with the columns of $\mathbf{D}^{\mathbf{t}}$ produces the zero row in **S**.
- Thus, the first column of **A** should be the last row of **V**, i.e.

$$\mathbf{DA} = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & ? \\ -1 & ? \\ -1 & ? \end{pmatrix} = \begin{pmatrix} 0 & ? \\ 0 & ? \\ 0 & ? \end{pmatrix}$$

Finding the Rest of $\mathbf{A}$

- Finding the last column of $\mathbf{A}$ is easy. Denote it $\mathbf{u}$.

$$\mathbf{DA} = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & u_1 \\ -1 & u_2 \\ -1 & u_3 \end{pmatrix} = \begin{pmatrix} 0 & \geq 1 \\ 0 & \geq 1 \\ 0 & \geq 1 \end{pmatrix}$$

- Multiplying each row of $\mathbf{D}$ with $\mathbf{u}$ should produce a positive number:

$$\begin{array}{ccccccc} 6u_1 & + & 4u_2 & + & 2u_3 & \geq & 1 \\ & & u_2 & - & u_3 & \geq & 1 \\ u_1 & & & + & u_3 & \geq & 1 \end{array}$$

- We find $\mathbf{u}$ to be e.g. $\mathbf{u} = (1, 1, 0)$.

$$\mathbf{A} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$

Computing $\mathbf{U}$

- Given a matrix $\mathbf{A}$, using a variant of the algorithm for echelon reduction, we can find a unimodular matrix $\mathbf{U}$ such that $\mathbf{A} = \mathbf{UT}$
- i.e.

$$\mathbf{A} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} = \mathbf{UT} = \begin{pmatrix} -1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} -1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$

Computing $\mathbf{L_U}$

- With this loop transformation matrix $\mathbf{U}$, we get the following new dependence matrix $\mathbf{D_U}$:
  $\mathbf{D_U} = \mathbf{DU}$
- i.e.

$$\mathbf{D_U} = \begin{pmatrix} 0 & 10 & 6 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} = \mathbf{DU} = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} -1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

- We don't actually need to compute $\mathbf{D_U}$ if we trust our compiler.
- The new loop $\mathbf{L_U}$ is constructed as explained before:
- A loop nest $\mathbf{L}$ is changed to a new loop nest $\mathbf{L_U}$ with loop index variables:

$$\mathbf{K} = \mathbf{IU}$$

- New array references and new loop bounds must be computed.
- We have already seen both of these two, but repeat them for convenience on the next two slides.

Recall: Computing the New Index Variables

- With

$$\left.\begin{array}{rcccl} \mathbf{p_0} & \leq & \mathbf{IP} & & \\ & & \mathbf{IQ} & \leq & \mathbf{q_0} \end{array}\right\} \tag{33}$$

$$\mathbf{I} = \mathbf{KU}^{-1} \tag{34}$$

We use Fourier-Motzkin elimination to find the loop bounds from

$$\left.\begin{array}{rcccl} \mathbf{p_0} & \leq & \mathbf{KU}^{-1}\mathbf{P} & & \\ & & \mathbf{KU}^{-1}\mathbf{Q} & \leq & \mathbf{q_0} \end{array}\right\} \tag{35}$$

- The bounds are found starting with k_1, k_2 etc.

Recall: New Array References

- All array references are rewritten to use the new index variables.
- Conceptually we could calculate, at the beginning of each loop iteration,
$$\mathbf{I} = \mathbf{K}\mathbf{U}^{-1}$$
and then use this vector $\mathbf{I}$ in the original references, on the form:
$$x[\mathbf{I}\mathbf{A} + \mathbf{a_0}]$$
- We don't do that of course and instead replace each reference with
$$x[\mathbf{K}\mathbf{U}^{-1}\mathbf{A} + \mathbf{a_0}]$$
- Here $\mathbf{K}\mathbf{U}^{-1}\mathbf{A} + \mathbf{a_0}$ can be calculated at compile-time.

The SUIF Compiler

- The SUIF compiler was developed at Stanford University.
- The project was lead by Monica Lam — who also invented modulo scheduling for software pipelining.
- Her project demonstrated that good parallelizing compilers not only must identify parallel loops, but at the same time perform cache optimizations to reduce communication between processors.

Parallelizing Compiler Challenges

- Due to synchronization and communication overhead, large computations must be identified that can execute in parallel.
- However, only identifying (or creating through transformations) parallelism is not sufficient.
- Early parallelizing compilers were not very good at optimizing both for:
    - parallelism, and
    - locality

  at the same time.
- This limited their success significantly.
- Due to shared memory and multiple levels of caches, multiprocessors make this complex.

Automatic Parallelization: Possible Portable Speedup

- As the SUIF compiler project showed, both parallelism and locality must be optimized.
- Without such compilers which can manage both, programmers must do it manually, which is expensive and time consuming.
- Furthermore, optimizing manually usually means tuning for a particular machine with its cache parameters.
- While performance can be good for a specific machine, there is a high risk the performance is not portable.
- The tuning needs to be repeated for other machines.
- That is another reason for using the best available parallelizing compiler.

Finding Coarse-Grain Parallelism

- Scalar analyses
- Array analyses
- Interprocedural analysis framework

Scalar Analyses

- Privatization of scalar variables
  - each thread gets its own variable
  - in addition to loop index variables, typically also ”temporary” variables used in loops which are defined and used in only one loop iteration
- Scalar reduction recognition (such as `sum += a[i]`)
- Constant propagation.
- Induction variable recognition and elimination.
- Motion of loop-invariant expressions out of loops.

Array Analyses

- Performs data dependence tests as explained earlier.
- The Fourier-Motzkin test is the most expensive and is performed last if the others fail to prove independence.
- SUIF can also privatize arrays.
- Recognition of reductions on array elements, i.e. parallel reductions.
- Some such reductions are even recognized on the form `a[b[i]]`.
- The latter is useful for sparse computations.

Interprocedural Analysis Framework

- Instead of the simpler solution of inlining is interprocedural dataflow analysis performed.
- Inlining does not scale to large programs and should not be used instead of proper interprocedural dataflow analysis.
- The dataflow information is expressed as a (mathematical) function of the (source code) function's arguments.
- When the calling contexts are sufficiently different, the framework selectively clones the called procedure, i.e. makes a copy which is tuned to the specific calling context.
- This approach gathers as much specific information as does inlining but consumes much less memory — both in the compiler and compiled executable.
- In the FORTRAN 77 application TURB3D from SPEC CPU95, loops with up to nine functions in 42 calls, and (if inlined) 86,000 lines were parallelized.
- The complete TURB3D benchmark is slightly more than 2000 lines.

Memory Optimization Issues

- Communication — true sharing misses
- Limited capacity — numerical applications often access huge amounts of data before reusing the same data which results in poor temporal locality.
- Limited associativity — if the data is mapped to the same cache locations there can be conflict misses
- Large cache block size — risks of false sharing misses
- The SUIF compiler tries to avoid these issues as explained shortly, and it attempts to hide the latency of the remaining misses through data prefetching.

Transformations

- The SUIF compiler analyses which parts of a large array each processor will access.
- For example, a 2D array where a processor accesses every nth row can be transformed into a 3D array where all accessed rows are contiguous in memory.
- Arrays can be transposed when that increases locality.
- By making a processor's data contiguous in memory, both true and false sharing misses are reduced.
- Loop tiling is also used for the resulting loops.

Performance

- The SUIF compiler set a world record for SPEC CPUfp95 using a machine with eight Alpha processors.
- Two of the benchmarks resulted in speedups of approximately 10 and 15 times. How can that happen?
- Of 18 applications, interprocedural analysis was essential for seven, and locality optimizations for three.