Optimizing for Parallelism and Locality
As we will see, parallelism and locality are two faces of the same coin. We consider applications that require heavy manipulation of array elements. Usually, scientific or engineering applications that require heavy manipulation of array elements.

Optimizing for Parallelism and Locality

We can compare older symmetric and multi-processors architectures with more modern Distributed Memory Machines.

There are two categories of code transformations:

- Affine partitioning and blocking
- Affine partitioning and blocking

We consider applications that require heavy manipulation of array elements.
Amdahl's Law

\[
\text{speedup} = \frac{1}{(1 - \frac{d}{f}) + \frac{d}{f}}
\]

\[
\text{speedup} = \frac{1}{1 - \frac{d}{f}} + \frac{d}{f}
\]

If \( f = 0.9 \), then still speedup can never be more than 2.

If \( f = 0.5 \), then the speedup can be more than 2.

Thus,

\[
\text{speedup} \geq 10.
\]
Loops are the most desirable targets of optimization.

Parallelism and Locality

A. Pnueli

Better than task-level parallelism

Loop-Level Parallelism

Loops are the most desirable targets of optimization.

A. Pnueli

Lecture 16: Parallelism and Locality

Honors Compilers, NYU, Spring, 2007

389
Locality comes in two flavors: temporal and spatial.

- **Temporal** — Multiple accesses to the same location are close in time on the same processor.
- **Spatial** — Accesses to spatially close locations are allocated to the same processor.

\[
\begin{align*}
&\text{for } i \in [0..n) \text{ do}
\end{align*}
\]  
\[
\begin{align*}
Z[i] &:= X[i] - Y[i] \\
Z[i] &:= Z[i] \times Z[i]
\end{align*}
\]  
\[
\begin{align*}
&\text{for } i \in [0..n) \text{ do}
\end{align*}
\]  
\[
\begin{align*}
Z[i] &:= X[i] - Y[i] \\
Z[i] &:= Z[i] \times Z[i]
\end{align*}
\]

This is a loop fusion transformation.
best performance. Decomposing the iterations into independent components leads to:

- Zeroing row-by-row in parallel:

  \[
  0 =: [i', i] Z \text{ do } (u \cdots 0) \in \ell \text{ do } ((I + d) \cdot q, u)\text{ min } d \cdot q \in \ell \text{ do } [NW/u] =: q
  \]

- Zeroing row-by-row:

  \[
  0 =: [i', i] Z \text{ do } (u \cdots 0) \in \ell \text{ do } i = q
  \]

- Zeroing column-by-column:

  \[
  0 =: [i', i] Z \text{ do } (u \cdots 0) \in \ell \text{ do } [N/u] =: 0
  \]
There are three relevant spaces:

**Affine Transform Theory**

- **Iteration Space** — Vectors of values assumed by the loop indices.
- **Data Space** — Set of array elements accessed.
- **Processor Space** — Set of processors in the system.

For the example:

```
for \( i \in [0..10] \) do
  \( Z[i+10] := Z[i] \)
```

There is no data dependence across iterations.

Honors Compilers, NYU, Spring 2007
Consider a program for matrix multiplication

\[ \lambda \ast X = Z \]

**Matrix Multiplication Example**

Lecture 16: Parallelism and Locality
If we have $p$ processors, we may partition the rows among them. But then the overall cost becomes

$$n^2 = c + p n^2$$

so the gain in computation speed is offset by the loss in communication speed.

If we partition the matrix into blocks of size $B$ and $B^2$, then the number of cache misses reduces to $2n^3/Bc$. If we partition the rows among them. But then the overall cost becomes

$$d u + \frac{u}{c} + \frac{u}{c + d n^2}$$

As the number of cache misses reduces, the second expression gets closer to $2n^3/Bc$. If we have processors, we may partition the rows among them. But then the overall cost becomes

$$d u + \frac{u}{c} + \frac{u}{c + d n^2}$$

The picture becomes more complex due to cache interference.

If we partition the matrix into blocks of size $B$ and $B^2$, then

$$c \geq B^2 \times B$$

The picture becomes more complex due to cache interference.
Iteration Spaces

Every nested loop structure defines a set of index vectors. The representation is not unique. For example, the code

\[
\begin{align*}
Z[3] &= 0 \\
\text{for } j = 0; & < 32; j++
\end{align*}
\]

is equivalent to

\[
\begin{align*}
Z[3] &= 0 \\
\text{for } j = 0; & < 100; j++
\end{align*}
\]

where

\[
\{ 0 \leq q + \mathcal{B} \cdot \mathcal{Z} \in \mathbb{Z} \}
\]
For example, consider the program

\[
0 = \left[ \begin{array}{c} j \\ell \end{array} \right] X \left[ \begin{array}{c} \ell \\ell \end{array} \right] \quad \text{for } \ell, j \in [0, 5] \quad \text{do for } \ell, j \in [0, 5] \quad \text{do for } \ell, j \in [0, 5]
\]

An example of an Iteration Space

The induced iteration space can be characterized as the set of inequalities representing the set of solutions:

\[
\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \leq \begin{bmatrix} \ell \\ \ell \end{bmatrix} + \begin{bmatrix} \ell \\ \ell \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}
\]
The inequality characterization of the iteration space does not determine the order in which these iterations are executed. The original program specifies a particular ordering which is consistent with lexicographic visits to the iteration points. The original program specifies a particular ordering which is consistent with lexicographic visits to the iteration points. This can be obtained by rewriting the loop as

\[
\text{for } j \in \{0, 7\} \text{ do for } i \in \{0, \min(5; j)\} \text{ do } \forall \ell \colon 0 \in [i, \ell] \forall \ell \colon 0 \in [i, \ell] \forall \ell \colon 0 \in [i, \ell] \forall \ell \colon 0 \in [i, \ell]
\]

where (a) displays the original access order, (b) presents a visitation order obtained by rewriting the loop as

\[
\text{for } \ell \in \{0, 7\} \text{ do for } i \in \{0, \min(5; \ell)\} \text{ do } \forall \ell \colon 0 \in [i, \ell] \forall \ell \colon 0 \in [i, \ell] \forall \ell \colon 0 \in [i, \ell] \forall \ell \colon 0 \in [i, \ell]
\]

In Fig. 11.4, we present various ordering of the program. A. Pnueli
Alg. 11.11

Fourier-Motzkin Elimination

Input: A polyhedron \( S \) over variables \( x_1, \ldots, x_n \).

Output: A polyhedron \( S_0 \) over variables \( x_1, \ldots, x_{n-1} \).

Method

1. For every pair of lower and upper bounds on such \( \mathbf{u} x \), create the new constraint \( \mathbf{u} x \mathbf{c} \mathbf{1} \geq \mathbf{T} \).

2. Reduce \( \mathbf{c} \mathbf{2} \mathbf{L} \) and \( \mathbf{c} \mathbf{1} \mathbf{U} \) by any common factor.

3. If the new constraint is unsatisfiable, then there is no solution. We may take \( S_0 \) to be the empty polyhedron.

4. Otherwise, \( S - S \) consists of \( \mathcal{C} - \mathcal{S} \) plus all the constraints generated in steps 1 and 2.

Let \( \mathcal{C} \) be all the constraints in \( \mathcal{S} \) involving \( \mathbf{u} x \). Let \( \mathcal{C} \) be all the constraints in \( \mathcal{S} \) involving \( \mathbf{u} x \).
Appling the F-M algorithm we obtain the trivial lower and upper bounds. Thus, the polyhedron is given by $\{ j \geq 0 \}$. Listing lower and upper bounds for $i$, we obtain $\{ j \geq 0 \}$. Eliminate $i$ from $\{ j \geq 0 \}$. A. Pnueli
Algorithm 11.13: Loop-Bounds Generation

Input: A convex polyhedron $S$ over variables $v_1, \ldots, v_n$.

Output: A set of lower bounds $\bar{\omega}$ and upper bounds $\bar{\Omega}$ for each $v_i$.

for $i \in [n]$ do

end for

Use Fourier-Motzkin to find the bounds in terms of $v_i$'s for $i \geq j$.

for $i \in [n]$ do

end for

for $i \in [1..n]$ do

end for

Remove any bounds in $L_{v_i}$ and $U_{v_i}$ implied by $S_0$.

Remove redundancies.

for $i \in [1..n]$ do

end for

for $i \in [1..n]$ do

end for

Add the remaining constraints of $S_0$ and $S_\bar{\omega}$ to $\bar{\omega}$.

Add the lower bounds on $\bar{\omega}$ in $S_\bar{\omega}$.

Add the upper bounds on $\bar{\omega}$ in $S_\bar{\Omega}$.

$S_\bar{\omega} := \bar{\omega} S$

$S_\bar{\Omega} := \bar{\Omega} S$

elimination of $v_i$ from $S_i$, using F-M.

$S_0 := \emptyset$

Honors Compilers, NYU, Spring, 2007
Example

Compute loop bounds for the polyhedron
Changing Axes

Sofar, we considered reordering iterations which only concerned the order by which the various indices changed.

However, we can also consider coordinate changes by arbitrary affine transformations. To the polyhedron associated with the transformation $i = j$, where $i$ is a new coordinate, applying the transformation $i = j - \ell$, this leads to $\ell \geq i \geq \ell + 5$.

Wishing to sweep the iteration space in the order $k, \ell$, we obtain the bounds $L_k : 0 \rightarrow 7$ and $U_k : 0 \rightarrow \infty$. This leads to $L_j : 0 \rightarrow \min(5 + k, 7)$ and $U_j : 0 \rightarrow \infty$.

For $k \in [0 : 7]$ do

$\{ \ell, i + 5 \} : \ell \wedge \ell \leq i \geq \ell + 5 \rightarrow \ell \\
\ell \wedge i \leq \ell + 5 \rightarrow \ell \wedge i \leq \ell \wedge i \leq 0$

Changing Axes

A.Pnueli
Each array access in a loop nest can be characterized by a tuple 

\[ (f, B, \mathcal{A}) = \mathcal{F} \]

where, for each \( i \), the reference \( \mathcal{F} \) of the loop is an affine expression of the loop indices and symbolic constants.

Examples:

Suppose \( i \) and \( j \) are the loop indices and \( n \) is a symbolic constant. Then \( [i \cdot n + 10]_j \) and \( [i \cdot 3 + 10]_j \) are affine expressions. If \( [i \cdot 3 + 10]_j \) is an affine expression, then so is \( [i \cdot 3 + 10]_j \). An array access in a loop is affine if

**Affine Accesses**
In many cases, it is possible to transform a non-affine program such as
\[
\{ j + 3 \mid 0 \leq i \leq n \} \quad \text{do} \quad 0 =: [j] Z \\
\{ 0 \mid 0 \leq i \leq n \} \quad \text{for} \quad i \in \mathbb{Z}^n + 2u + 2
\]
into an affine program such as
\[
0 =: [? * 2 + u] Z \\
\text{for} \quad i \in \mathbb{Z}^n + 2u + 2
\]

**Transformation Into Affine Programs**
DataReuse

Analysis of a for loop can lead to two types of optimizations:

DataReuse

Types of Reuse

Reordering transformations:

- Analyze dependence between various accesses to the same data. This identifies the range of allowed reorderings of the same data.

- Group together iterations which perform multiple accesses to the same or spatially close data.

- Temporal reuse: the same exact location is repeatedly referenced. It is spatial if the locations come from different static accesses, then it is self reuse. If they come from the same static access.

- Group reuse: access to the same data. This identifies the range of allowed reorderings of the same data.

- Self reuse: multiple access of affine loops can lead to two types of optimizations.
Consider a loop nest of depth $d$ with accesses which access data of dimension $r<d$. Obviously, the same data must be referenced repeatedly.

The dimensionality $r$ of the data accessed is determined by the rank of the matrix $\mathbf{A}$ in the affine access function $\mathbf{A}\mathbf{x} + \mathbf{b}$ such that the transformed affine access function $\mathbf{A}\mathbf{x} + \mathbf{b}$ should depend only on the first $r$ indices of the loop nest.

We should apply an affine transformation to the iteration space, such that the transformed affine access function $\mathbf{A}\mathbf{x} + \mathbf{b}$ should depend only on the first $r = \text{rank}(\mathbf{A})$ indices of the loop nest.

For example, in the third example of Figure 1.19, we should reorganize the loop to be of the form:

$$\{ \cdots [I + \mathbf{c}] \cdots \} \mathbf{x} \cdots \mathbf{x}$$
This is the case that a single static reference of the form
\[
\sum_i \sum_j A[i,j] = 0
\]

refers to data elements which are spatially close.

Spatial Self-Reuse

Here, we transform again the access function so that it will depend on the indices of the loop nest. Furthermore, we make an effort that the dimension which guarantees closest proximity (columns in C, rows in Fortran) will depend on the \( i \)’th index. Thus, we will transform the C-loop into

\[
\sum_i \sum_j A[i,j] = 0
\]
Group Reuse

We consider group reuse for the case that the static references are of the form $\vec{F}_i + \vec{f}_1$ and $\vec{F}_i + \vec{f}_2$ (i.e., same $F$).

For example, the program demonstrates good reuse and should not be modified.

```
for i ∈ [1..n] do for j ∈ [1..n] do Z[i,j] := Z[i-1,j]
```
Array Data-Dependence Analysis

Consider two static accesses to the same array in possibly different loops; associated with these accesses are data dependent if

\[ \langle p, e, f, b, \delta \rangle = f = \langle p, e, f, b, \delta \rangle \]

In particular, we may ask the question of dependence between a static reference and itself. In this case, case we should add the requirement that \( i \neq i' \).

The accesses are data dependent if

\[ \langle p, e, f, b, \delta \rangle = f = \langle p, e, f, b, \delta \rangle \]

There exist \( i \in \mathbb{Z}_d \) and \( i' \in \mathbb{Z}_d \) such that

At least one of them is a write.

At least one of them is a write.

At least one of them is a write.
Consider the loop:

\[ \text{for } i \in [1..10] \text{ do } Z[i] = \ldots \]

Analyzing dependences, we obtain:

- Dependence between \( Z[i] \) and itself.
- Dependence between \( Z[i-1] \) and \( Z[i] \) does not exist.

These exist for all pairs \( i, j \) such that \( 2 \leq i + j \leq 9 \) and \( i < j < 10 \).

\[ Z[i] = Z[i-1] \]

We need not worry about self-dependence of \( Z[i] \) because this is a reading reference.

Example
The data dependency problem requires solving an integer linear program. The approach presented here consists of three parts:

1. If this fails, apply an ILP solver that uses a branch-and-bound approach based on F-M.

2. Use a set of heuristics to deal with the inequalities.

3. Apply a GCD (greatest common divisor) test which checks if there exists an integer solution to the equalities. If there exists no solution, then there are no dependencies. Otherwise, use the equalities to substitute some of the variables by expressions in terms of the other variables.

The approach presented here consists of three parts:

1. The heuristics for solving large ILP problems.

2. This is an NP-complete problem, but there exists many variables. This is an NP-complete problem, but there exists many variables.

3. The data dependency problem requires solving an integer linear program.
A. Pnueli

The GCD Test is based on Claim 8. (11.32)

The Linear Diophantine Equation

Claim 8. (11.32)

The GCD Test

 Otherwise, replace some of the variables by expressions in terms of others. Otherwise, replace some of the variables by expressions in terms of others.

We apply a Gaussian-elimination like algorithm to express some of the variables in terms of the others. Subject every generated equation to the GCD test. If any of these fail, there is no solution. Otherwise, replace some of the variables by expressions in terms of the others.

has an integer solution for } x1, x2, \ldots x_n \text{ iff } \gcd(a_1, a_2, \ldots, a_n) \text{ divides c.}

\begin{align*}
c = u_1 x_1 + u_2 x_2 + \cdots + u_n x_n
\end{align*}

Lecture 16: Parallelism and Locality

Honors Compilers, NYU, Spring, 2007
Consider the equations
\[ x^2 y + z = 0 \]
\[ 3x + 2y + z = 5 \]
Both equations pass the GCD test. However, subtracting the first equation from the second, we obtain
\[ 2x = 4y + 5 \]
which fails the GCD test.

Example
We will consider fast heuristics for solving the inequalities part of an ILP problem.

**Independent-Variables Test**

Assume that there are one or more variables \( t \) whose constraints are constant lower bounds

\[
0 < \frac{u_p}{u_L} \leq \frac{u}{u_L} \leq \frac{t_p}{t_L} \leq \frac{t}{t_L} \leq \frac{c_1}{c_1} \leq \cdots \leq \frac{c_m}{c_m} \leq \frac{c}{c} \leq 0
\]

Furthermore, with no loss of generality, we can take \( t \) to be

\[
\text{max}(T, \frac{u_p}{u_L}, \frac{u}{u_L}, \frac{t_p}{t_L}, \frac{t}{t_L}, \cdots)
\]

Then, the system is solvable for \( t \) if

\[
0 < \frac{u_p}{u_L} \leq \frac{t_p}{t_L} \leq \frac{t}{t_L} \leq \frac{c_1}{c_1} \leq \cdots \leq \frac{c_m}{c_m} \leq \frac{c}{c} \leq 0
\]
Consider the following set of inequalities:

\[ \begin{align*}
0 & \geq t_1 \\
0 & \geq t_2 \\
0 & \geq 10 \\
11 & \geq t_1 + t_2 \\
11 & \geq t_1 \\
11 & \geq t_2 \\
11 & \geq 10 \\
10 & \geq t_1 \\
10 & \geq t_2 \\
11 & \geq 0 \\
10 & \geq 0 \\
10 & \geq 0 \\
10 & \geq 0 \\
0 & \geq 0 \\
0 & \geq 0
\end{align*} \]

which can be simplified to

\[ \begin{align*}
10 & \geq t_1 + t_2 \\
10 & \geq t_1 \\
10 & \geq t_2 \\
0 & \geq 0 \\
0 & \geq 0 \\
0 & \geq 0 \\
0 & \geq 0 \\
0 & \geq 0
\end{align*} \]

This is obviously unsolvable since the lower bound of \( t_2 \) is bigger than the upper bound of \( t_1 \).
Assumethattheonlylowerboundon $v_i$ is of theform $d_0$ and allupperbounds on $v_i$ are of theform $c_1v_i + \ldots + c_nv_i+1+v_i+1+c_{i+1}v_i+1+\ldots+c_nv_n$.

If there is no lower bound, we can take $v_i = -\infty$. Assume that the only lower bound on $v_i$ is of theform $v_i = c_0 + c_1v_i + \ldots + c_nv_n$. Then we can replace $v_i$ by $v_i = c_0v_i + \ldots + c_nv_i+1+v_i+1+c_{i+1}v_i+1+\ldots+c_nv_n$.

Consecquently, we can take $v_i = \frac{d_0}{d_i}$ and obtain $\frac{d_0}{d_i} + v_i + \ldots + v_i+1+v_i+1+c_{i+1}v_i+1+\ldots+c_nv_n$.

\section{Acyclic Test}

Considertheinequalities $1v_1, v_2 \geq 10, 1v_1 \geq v_3 \geq 1v_3 \geq 1v_2 \geq 1v_2$.

\begin{itemize}
  \item $v_1$ is lower-bounded by 1 and $v_2$ is upper-bounded by 10.
  \item Assume that the only lower bound on $v_i$ is of theform $v_i \geq d_0$.
  \item If there is no lower bound, we can take $v_i = -\infty$.
\end{itemize}
Consider the case that a subset of the variables has only constraints of the form
\[ v_i + c \leq v_j + c \]

Let \( L_i \) and \( U_i \) be the lower and upper bounds for \( v_i \) respectively.

We construct a graph whose nodes are the variables plus a special node labeled 0 (representing the constant 0). For each constraint
\[ v_i + c \leq v_j + c \]
we construct a graph node labeled 0 \( v_i + c \), and from node labeled 0 to label \( v_j + c \) we draw an edge. Every constraint creates a self-loop.

Consider the case that a subset of the variables has only constraints of the form
\[ v_0 \leq v_i + c \leq v_0 \]

If the process of constraint tightening creates a self-loop by constraint \( c_i + c_j \), then replace \( c_i + c_j \) with \( c_i + c_j + (c_i + c_j) \).

If \( c_i + c_j > v_0 \), then the constraints imply
\[ v_0 \leq v_i \leq v_0 \]

For every \( i, j, k \), the constraints are represented as
\[ v_i + c_i + c_j \leq v_j + c_i + c_j \leq v_j + c_i + c_j \]

and
\[ v_i + c_i + c_j \leq v_j + c_i + c_j \leq v_j + c_i + c_j \]

Consider the case that a subset of the variables has only constraints of the form
\[ v_0 \leq v_i + c \leq v_0 \]

If the process of constraint tightening terminates, it is possible to read a solution from the graph.

Otherwise, when with negative weight, the problem is inconsistent. Otherwise, when
\[ c_i + c_j + c_k \]

loop residue test