Chapter 2

Dependence Analysis

As mentioned earlier extraction of parallelism from sequential programs should not violate semantics of the sequential program. Thus to preserve the semantics an extensive dependence analysis has to be done. Informally dependence analysis consist of capturing dependence between statements of a program which need to be preserved while extracting parallelism. For example, if statement S_1 writes a variable which is later used by statement S_2, then, S_2 cannot execute before S_1, since otherwise S_2 will end up using old copy of the variable. Thus the dependence structure of a program is determined by the way memory locations are referenced i.e. read and written and the way control flows during the sequential execution of the program. The dependence of the first type is called data dependence while of the later type is called control dependence. In our work we will be more concerned with data dependence rather than control dependence. In our work we have considered a nest of '`for`' loops as shown in Fig. 5.9 . The column vector $\bar{i}$ called the index point is $\bar{i} = (i_1, i_2, i_3, \ldots, i_n)^T$. $S_1, S_2, S_3, \ldots, S_n$ are the assignment statements typically involving array subscripts. Execution of the nest

```
for( i1 = l1; i1 ≤ u1; i1 + + )
  for( i2 = l2; i2 ≤ u2; i2 + + )
    for( i3 = l3; i3 ≤ u3; i3 + + )
      for( in = ln; in ≤ un; in + + )
      {
          S1(ī );
          S2(ī );
          S3(ī );
            ⋮
          Sm(ī );
      }
```

Figure 2.1: **Showing the nest of '`for`' loop**

of '`for`' loops generates index points that are ordered lexicographically. The index set of the loop is $I = \{(i_1, i_2 i_3, \ldots, i_n)^T : l_1 \leq i_1 \leq u_1, \ldots, l_n \leq i_n \leq u_n\}$.

Thus we say statement S_2 depends on statement S_1 denoted by $S_1 \delta S_2$ if there exists an instance $S_1^{'}$ of S_1 and an instance $S_2^{'}$ of S_2 and a memory location M such that

1. Both $S_1^{'}$ and $S_2^{'}$ reference M, and at least one of those reference is a write.
2. In the serial execution of the program $S_1^{'}$ is executed before $S_2^{'}$.
3. In the same execution M is not written between the time $S_1^{'}$ finishes and the time $S_2^{'}$ starts.

We define three types of dependence based upon the types of the two references to M. Statement S_2 is

i) flow–dependent (denoted by $S_1 \delta^f S_2$) on S_1 , if $S_1^{'}$ writes M and then $S_2^{'}$ reads it.

ii) anti–dependent (denoted as $S_1 \delta^a S_2$) on S_1 , if $S_1^{'}$ reads M and then $S_2^{'}$ writes it.

iii) output–dependent (denoted as $S_1\delta^o S_2$) on S_1 , if $S_1^{'}$ writes M and then $S_2^{'}$ writes it.

We define, for a statement S_1 , $IN(S_1)$ as the set of variables in the read set of S_1 i.e. all the variables that are read by S_1. Similarly, we define $OUT(S_1)$ as the set of variables in the write set of S_1 i.e. all the variables that are written by S_1 .

Based on the above definitions we have

i) $S_1\delta^f S_2$, if $\ni x$ such that $x \in OUT(S_1)$ and $\ni y$ such that $y \in IN(S_2)$.

ii) $S_1\delta^a S_2$ if if $\ni x$ such that $x \in IN(S_1)$ and $\ni y$ such that $y \in OUT(S_2)$.

iii) $S_1\delta^o S_2$ if $\ni x$ such that $x \in OUT(S_1)$ and $\ni y$ such that $y \in OUT(S_2)$.

Example 2.1: Consider the nest of 'for' loops shown below.

$$
\begin{array}{l}
\texttt{for}\,(i_1 = 0; i_1 < 100; i_1 + +) \\
\quad \texttt{for}\,(i_2 = 0; i_2 < 100; i_2 + +) \\
\quad \{ \\
\qquad\quad S_1 : \; a[i_1][i_2] = \\
\qquad\quad S_2 : \qquad\quad = a[i_1 - 1][i_2 - 2]; \\
\quad \}
\end{array}
$$

$S_1\delta^f S_2$ since $S_1(i_1^{'}, i_2^{'})$ writes variable $a[i_1^{'}][i_2^{'}]$ and later $S_2(i_1^{'}+1, i_2^{'}+2)$ reads it. However, if we have S_1 and S_2 as

$S_1 : \qquad\quad = a[i_1][i_2];$

$S_2 : \; a[i_1 - 1][i_1 - 2] =$

then S_2 is anti–dependent on S_1 since $S_1(i_1^{'}, i_2^{'})$ reads variable $a[i_1^{'}][i_2^{'}]$ and later $S_2(i_1^{'} + 1, i_2^{'} + 2)$ writes it. In between no other iteration writes $a[i_1^{'}][i_2^{'}]$.

Let S_1 and S_2 above refer to the following statements

$S_1: \quad a[i_1][i_2] =$

$S_2: \quad a[i_1 - 1][i_1 - 2] =$

For the above statements we have S_2 output–dependent on S_1 , since both $S_1(i_1^{'}, i_2^{'})$ and $S_2(i_1^{'} + 1, i_2^{'} + 2)$ write the variable $a[i_1^{'}][i_2^{'}]$ and no other iteration in between modifies $a[i_1^{'}][i_2^{'}]$.

Suppose statement S_2 depends on S_1. Let $S_1(\overline{i})$ and $S_2(\overline{j})$ denote a pair of instances such that they satisfy the dependency conditions. Thus $S_1(\overline{i})$ must be executed before $S_2(\overline{j})$. This relation is captured by a Dependence Vector denoted by $\overline{d} = \overline{j} - \overline{i}$. Thus the Dependence Vector in Example 2.1 is $\overline{d} = [1 \;\; 2]^T$. An algorithm may contain many such dependence vectors. Hence we define the dependence matrix as $D^{n \times m} = \left[\overline{d}_1 \; \overline{d}_2 \; \overline{d}_3 \ldots \overline{d}_m\right]$ where $\overline{d}_1, \overline{d}_2, \overline{d}_3, \ldots, \overline{d}_m$ are the m dependence vectors. Some programs have a dependence matrix all of whose entries are constant. Such programs are said to have a Uniform Dependence Structure as defines below.

Definition : Uniform Dependence Structure

A program is said to have a Uniform Dependence Structure if the entries of the dependence matrix are constant, i.e., independent of the loop indexing variables.

A program with uniform dependence structure is much easier to parallelise than one without uniform dependence structure. For such programs concept of direction vector is used. The direction vector $\overline{v} = [\overline{v}_1 \; \overline{v}_2 \; \overline{v}_3 \ldots \overline{v}_m]$ is defined as

$$v_i : i = 1 \ldots n \begin{cases} < & \text{if } d_i > 0 \\ = & \text{if } d_i = 0 \\ > & \text{if } d_i < 0 \end{cases}$$

In our work we have considered sequential programs with a Uniform Dependence

structure.

Distance vectors (dependence vectors are of whose entries are constant) are preferable over direction vectors since they encompass the exact dependence structure of the algorithm and thus are more suitable for extraction of more parallelism.

We say that a loop carried dependence exists at the i^{th} loop in a loop nest if at least one of the entries in the i^{th} row of a dependence matrix is non–zero. Loop carried dependence prohibits conversion of i^{th} loop into a '`for_all`' For such a loop '`for_across`' with synchronisation between iterations can be used. If all the entries at i^{th} row of a dependence matrix D are zero, then the i^{th} loop can be converted into a '`for_all`' loop for independent execution on processors of a MIMD machine. Direction vectors are useful in determining whether a loop carried dependence exists and also whether a loop interchange is legal.

2.1 Tests for Dependence

As mentioned earlier the first step in program transformation for parallel processing is detection of a data dependence. Informally a dependence test involves testing if two indexed elements of an array would represent the same memory location. Loop indices being integer variables, the dependence problem reduces to finding integer solutions to a set of Diophantine equations subject to a set of inequality constraints. The dependence problem becomes more difficult if the array indices are non–linear functions of the index variables. However, in real programs the array indices are usually linear functions of the index variable. Hence we deal with dependence analysis in which array indices are linear functions of the index variables.

2.1.1 Diophantine Analysis

Diophantine analysis involves finding integer solutions of a linear equation. Let a and b be integer numbers $(b \neq 0)$. We say that b divides a, denoted by $a|b$, if and only if there is an integer x such that $a = bx$. A linear Diophantine equation is of the form $\sum_{k=1}^{n} a_k x_k = c$, where $n \geq 1$, c, a_k are integers, x_k are integer variables. A Diophantine equation has a solution if and only if $GCD(a_1, a_2, a_3, \ldots, a_n) \mid c$. For a detailed discussion on Diophantine equations the reader is referred to. We consider solution to a Diophantine equation involving two variables.

Consider a Diophantine equation shown below.

$$ax_1 + bx_2 = c$$

Let $g = GCD(a, b)$. The solution to this Diophantine equation is of the form

$$x_1 = \left(\frac{b}{g}\right) t + \left(\frac{c}{g}\right) x_{10}$$

$$x_2 = -\left(\frac{a}{g}\right) t + \left(\frac{c}{g}\right) x_{20}$$

where $g = ax_{10} + bx_{20}$.

Example

Consider a '`for`' loop shown below.

$$
\begin{array}{l}
\texttt{for}\,(i = 1; i <= 101; i++) \\
\{ \\
\quad S_1 : a[2 * i] = \\
\quad S_2 : \qquad = a[3 * i + 198]; \\
\}
\end{array}
$$

For a dependence to exist $S_1(i_1)$ and $S_2(i_2)$ must refer the same memory location. Hence we get the dependence equation $2*i_1 = 3*i_2+198$ or $2*i_1-3*i_2 = 198$. Since $g = GCD(2,-3) = 1$ we get the solution to the above equation as

$$i_1 = -3t + 198i_{10}$$

$$i_2 = -2t + 198i_{20}$$

Since $2i_{10} - 3i_{20} = 1$ let $i_{10} = 2,\ i_{20} = 1$. Hence we get

$$i_1 = -3t + 396$$

$$i_2 = -2t + 198$$

The constraints on i_1 and i_2 become constraints on t . Hence

$$1 \leq -3t + 396 \leq 101$$

or

$$99 \leq t \leq 131$$

Similarly

$$1 \leq -2t + 198 \leq 101$$

or

$$49 \leq t \leq 98$$

Since there is a contradiction $t \leq 98$ and $99 \leq t$, the Diophantine equation does not have a solution in the region bounded by the iteration space. Hence the pair of references are not data dependent.

2.1.2 GCD Test

Let the linear Diophantine equation resulting from equating the array indices be of the form

$$f(x) = a_1x_1 + a_2x_2 + a_3x_3, \ldots, a_nx_n = c$$

where $x_i : 1 = 1, \ldots, n$ are integer variables $a_i : i = 1, \ldots, n$ and c are constants. Integer solution to the above equation exists if $GCD(a_1, a_2, a_3, \ldots, a_n) \mid c$. Consider the '`for`' loop shown below

```
for (i = 1; i <= 100; i++)
{
    S1 :   a[2 * i - 4] =
    S2 :          = a[3 * i - 1]
}
```

To test the dependence between S_1 and S_2 we get the dependence equation

$2i_1 - 4 = 3i_2 - 1$ or $2i_1 - 3i_2 = 3$. Since $GCD(2, -3) = 1$ and $1|3$ integer solution to this equation exists and thus dependence between S_1 and S_2exists.

2.1.3 Banerjee's Inequality Test

GCD test is an inexact test because even if the test succeeds the integer solution may be out of bounds of the index space and hence dependence may not exist. Hence Banerjee's Inequality Test is applied to check whether the solution lies within the bounds of the index space. Let the linear Diophantine equation whose solution needs to be checked be of the form

$$f(x) = a_1x_1 + a_2x_2 + a_3x_3, \ldots, a_nx_n = c$$

where $x_i : 1 = 1, \ldots, n$ are integer variables $a_i : i = 1, \ldots, n$ and c are constants. This equation results from setting the array indices equal in a pair of array references. Let $R \subset Z^n$ be a rectangular iteration space as defined below.

$$\{(x_1, x_2, x_3, \ldots, x_n) \in Z^n : l_1 \leq x_1 \leq u_1, l_2 \leq x_2 \leq u_2, \ldots, l_n \leq x_n \leq u_n\}$$

Then the maximum and minimum values of f in R are obtained from the formulas

$$f_{min} = \sum_{k=1}^{n} (a_k^+ l_k - a_k^- u_k)$$

and

$$f_{max} = \sum_{k=1}^{n} (a_k^+ u_k - a_k^- l_k)$$

where a_k^+ stands for the positive part of a_k

$$a_k^+ = \begin{cases} a_k,\ a_k \geq 0 \\ 0\, a_k < 0 \end{cases}$$

and a_k^- stands for the negative part of the real number.

$$a_k^- = \begin{cases} -a_k,\ a_k \leq 0 \\ 0\, a_k > 0 \end{cases}$$

If c does not satisfy $f_{min} \leq c \leq f_{max}$ then there is no solution in R and hence dependence does not exist. If c does lie within this range there may or may not be solution in R . In that case assume that there is such a solution.

Example

Chapter 3

Loop Transformations

Over the last two decades or so many techniques for extraction of parallelism have been developed. In this chapter we survey these techniques of loop transformation for extraction of parallelism. The techniques typically transform a sequential nest of '`for`' loops in to a form suitable for execution on the underlying parallel architecture. Loop distribution, for example, is suitable for vectorisation of a '`for`' loop nest. Conversion of a '`for`' loop into a '`for_all`' is suitable for execution on a machine with multiprocessor architecture. A vectorising or a parallelising compiler has these techniques integrated for automatic generation of parallel code.

'`for_all`' and '`for_across`' code conversions alongwith the synchronisation primitives required for synchronisation between iterations of a '`for_across`' loop have been presented in this chapter. Loop transformation techniques such as loop distribution, loop fusion, loop interchange have been discussed next. We finally discuss unimodular transformations of a nest of '`for`' loops. Unimodular transformations is an important mathematical tool for program transformations. It

is shown how transformation techniques such as loop skewing, loop permutationand loop reversal can be represented by unimodular transformations.

3.1 'for_all' and 'for_across' code conversion

Consider the nest of 'for' loops shown below.

$$\begin{aligned}&\texttt{for}\,(\ \ i=0; i<N_1; i++)\\ &\quad\texttt{for}\,(j=0; j<N_2; j++)\\ &\quad\quad a[i][j]=a[i][j-2];\end{aligned}$$

The dependence vector for this loop nest is $[0 \ \ 2]^T$. Since loop carried dependence does not exist in the outermost dimension, the outermost loop can be converted into a 'for_all' loop with no synchronisation between iterations.

$$\begin{aligned}&\texttt{for\_all}\,(i=0; i<N_1; i++)\\ &\quad\texttt{for}\,(j=0; j<N_2; j++)\\ &\quad\quad a[i][j]=a[i][j-2];\end{aligned}$$

Thus each processor executes the innermost loop for a fixed value of i. However, if we change the assignment statement to $a[i][j] = a[i-2][j]$ then the dependence vector becomes $[2 \ \ 0]^T$ and the innermost loop becomes a 'for_all' loop.

$$\begin{aligned}&\texttt{for}\,(i=0; i<N_1; i++)\\ &\quad\texttt{for\_all}\,(j=0; j<N_2; j++)\\ &\quad\quad a[i][j]=a[i-2][j]\end{aligned}$$

Thus the innermost loop can be executed in parallel but synchronisation is required between iterations of the outer loop. For a vector processor the above nest of loops reduces to

$$a[0 \ldots N_1 - 1][0 \ldots N_2 - 1] = a[-2 \ldots N_1 - 3][0 \ldots N_2 - 1]$$

If loop carried dependence exists in a loop then it cannot be converted into a '`for_all`'. However, in this case it can be converted into a '`for_across`' loop with use of synchronisation primitives such as random synchronisation, critical sections, pipelining and barriers which are discussed in the following section.

3.2 Synchronisation Primitives

3.2.1 Critical Sections

Consider the '`for_across`' loop shown below

```
for_across (i = 2; i <= N; i++)
{
    begin critical section
    S1 :    a[i] = a[i − 1] ∗ b[i] + c[i];
    end critical section
    S2 :    c[i] = a[i] + c[i];
    S3 :    d[i] = a[i] ∗ 2;
    S4 :    e[i] = d[i] + c[i];
}
```

The dependence of S_1 on S_1 with distance $[1]$ prevents '`for_all`' execution of the loop. Hence we put the statement in a critical section. Each iteration is executed on a distinct processor. However, all processors execute S_1 in a serial fashion and then execute the other part of the loop concurrently.

While using critical sections the goal should be to insert as few critical sections as necessary and as small in size as possible since the speedup is limited by the size of the largest critical section. Generally the statements involved in dependence need to be put in a critical section.

3.2.2 Random Synchronisation

Consider the '`for`' loop shown below

`for` $(i = 0; i < N; i++)$

{

$S_1 : \quad a[i] = b[i] + 2;$

$S_2 : \quad d[i] = a[i]/d[i];$

$S_3 : \quad c[i] = 2 * d[i] + a[i-1];$

}

In this loop we have $S_1 \delta^f S_3$. Thus $S_3(i)$ can be executed only after $S_1(i-1)$ has completed. We get the following '`for_across`' code using `wait` and `post` primitives.

```
for_across (i = 0; i < N; i++)
{
    S1 :  a[i] = b[i] + 2;
    post (ASYNC, i);
    S2 :  d[i] = a[i]/d[i];
    wait (ASYNC, i − 1);
    S3 :  c[i] = 2 ∗ d[i] + a[i − 1];
}
```

The i^{th} iteration being executed on a processor waits for a `post` from the $i - 1^{th}$ iteration being executed on a different processor thus achieving synchronisation for satisfying the dependence $S_1 \delta^f S_3$. Random synchronisation has the advantage of flexibility.

3.2.3 Pipelining

In pipelining the '`for`' loop is divided into segments of code such that source and sink of a dependence lie within the same segment.

Consider the `for` loop shown below.

```
for (i = 0; i < N; i++)
{
    S1 :  a[i] = b[i] + c[i − 1];
    S2 :  c[i] = a[i − 1] + c[i];
    S3 :  d[i] = c[i] ∗ 2;
    S4 :  e[i] = d[i] + c[i − 1];
    S5 :  f[i] = e[i] + f[i];
    S6 :  g[i] = f[i] ∗ 2 + d[i];
}
```

Using pipelining for synchronisation we get the following '`for_across`' loop.

```
for_across (i = 0; i < N; i++)
{
    seg 1 :  S1 :  a[i] = b[i] + c[i − 1];
    seg 1    S2 :  c[i] = a[i − 1] + c[i];

    seg 2 :  S3 :  d[i] = c[i] ∗ 2;
    seg 2 :  S4 :  e[i] = d[i] + c[i − 1];

    seg 3 :  S5 :  f[i] = e[i] + f[i];
    seg 3 :  S6 :  g[i] = f[i] ∗ 2 + d[i];
}
```

The synchronisation is achieved because execution of `seg p` of i^{th} iteration

will finish before execution of the same segment for any subsequent iteration.

3.2.4 Barrier synchronisation

This synchronisation strategy handles only forward references. In this strategy a loop is divided into segments but all iterations must complete execution of a segment before execution of the next segment is started.

Consider the '`for_across`' loop shown below.

```
for_across (i = 0; i < N; i++)
{
    S1 :  a[i] = b[i] + 2;
   barrier
    S2 :  c[i] = a[i − 1] + c[i];
    S3 :  d[i] = a[i] + 2;
     barrier
    S4 :  e[i] = d[i] + c[i − 1];
    S5 :  f[i] = e[i] + f[i];
}
```

Since $S_1 \delta^f S_2$ and $S_2 \delta^f S_4$ barriers have been used after statements S_1 and S_3 to satisfy the above dependencies.

3.3 Transformation Techniques

3.3.1 Loop Coalescing

This method transforms several nested concurrent loops into a single concurrent loop. Thus scheduling of the loops is easier.

Consider the nest of '`for_all`' loops shown below

$$
\begin{aligned}
&\texttt{for\_all}\,(i = 1; i <= N_1; i++)\\
&\quad\texttt{for\_all}\,(j = 1; j <= N_2; j++)\\
&\quad\quad\texttt{for}\,(k = 1; k <= N_3; k++)\\
&\quad\quad\quad a[i][j] = a[i][j] + b[i][k] * c[k][j];
\end{aligned}
$$

The outer `for_all` loops can be coalesced into a single `for_all` loop by linearising the two dimensional iteration space as shown below

$$
\begin{aligned}
&\texttt{for\_all}\,(ij = 1; ij <= N_1 N_2; ij++)\\
&\quad\texttt{for}\,(k = 1; k <= N_3; k++)\\
&\quad\quad a[ij/N_2][\lceil ij/N_1 \rceil] = a[\lceil ij/N_2 \rceil][\lceil ij/N_1 \rceil] = b[\lceil ij/N_2 \rceil][k] * c[k][\lceil ij/N_1 \rceil];
\end{aligned}
$$

3.3.2 Loop Distribution

Nested loops can be vectorised by distributing the outermost loop and vectorising each of the resulting loops or loop nests. Thus this loop transformation method is very much suitable for vector machines.

Consider the loop nest shown below

```
for (i = 1; i <= N; i++)
{
    b[i][1] = 0;
    for (j = 1; i <= M; j++)
    a[i] = a[i] + b[i][j] * c[i][j];
    g[i] = f[i] + a[i];
}
```

The i loop is distributed to three copies, separated by the nested j loop from the assignment to arrays b and g.

```
b[1 : N][1] = 0;
for (i = 1; i <= N; i++)
    a[i] = a[i] + b[i][1 : M] * c[i][1 : M];
g[1 : N] = f[1 : N] + a[1 : N];
```

Consider the loop as shown below

```
for (i = 0; i < N; i++)
{
    S1 :  a[i + 1] = b[i − 1] + c[i];
    S2 :  b[i] = a[i] * k;
    S3 :  c[i] = b[i] − 1;
}
```

The dependencies for this loop are $S_1\delta^f S_2,\ S_2\delta^f S_1,\ S_1\delta^a S_3$ and $S_2\delta^f S_3$.

After applying loop distribution we get the following code.

```
for (i = 0; i < N; i++)
{
    S1 :  a[i+1] = b[i-1] + c[i];
    S2 :  b[i] = a[i] * k;
}
c[0...N-1] = b[0...N-1] - 1;
```

3.3.3 Node splitting

The data dependence cycle in a loop which prevents loop vectorisation can sometimes be broken by node splitting. Consider the following loop.

```
for (i = 1; i <= N; i++)
{
    S1 :  d[i] = a[i-1] + a[i+1];
    S2 :  a[i] = b[i] + c[i];
}
```

We have the dependence cycle $S_1\delta^a S_2,\ S_2\delta^f S_1$ which prevents vectorisation. However, we can split statement S_1 into two parts and apply statement reordering.

$$
\begin{aligned}
&\texttt{for}\,(i = 1; i <= N; i++)\\
&\{\\
&\qquad S_{1a}: \quad x[i] = a[i+1];\\
&\qquad S_{2}: \quad a[i] = b[i] + c[i];\\
&\qquad S_{1b}: \quad d[i] = a[i-1] + x[i];\\
&\}
\end{aligned}
$$

This loop does not have a dependence cycle and thus can be vectorised as follows.

$$
\begin{aligned}
S_{1a}: &\quad x[1:N] = a[2:N+1];\\
S_{2}: &\quad a[1:N] = b[1:N] + c[1:N];\\
S_{1b}: &\quad d[1:N] = a[0:N-1] + x[1:N];
\end{aligned}
$$

3.3.4 Statement substitution

If the dependence graph for a loop has multi–statement cycles, then statement reordering cannot be used to vectorise the loop. In this case statement substitution can be used to reduce the dependence cycle to a single statement cycle.

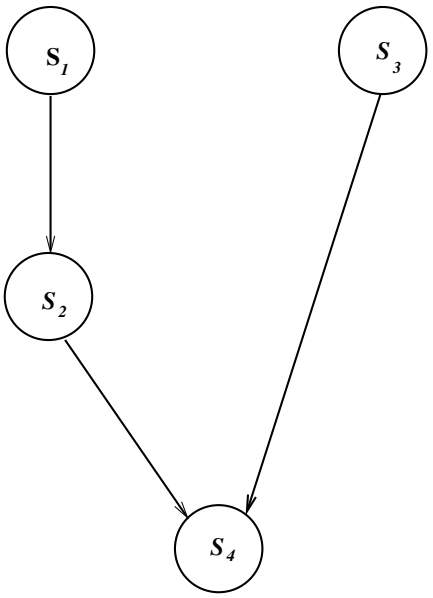
If we have two assignment statements, S_μ and S_ν, where the left hand side variable of S_μ is 'a' and 'a' is used on the right hand side of S_ν and $S_\mu \delta S_\nu$, then we may be able to substitute the right hand side expression of S_μ for the reference to 'a' in S_ν. The dependence graph for the modified program will also change. The dependence $S_\mu \delta S_\nu$ will be removed from the dependence graph. Then dependence relation $S_\omega \delta S_\nu$ will be added for all statements S_ω such that

$S_\omega \delta S_\mu$ was in the original dependence graph. Similarly, relations $S_\nu \delta^a S_\gamma$ will be added for all statements such that $S_\mu \delta^a S_\gamma$ was in the original dependence graph.

Consider the example shown below.

$$
\begin{aligned}
S_1: &\quad a = 1; \\
S_2: &\quad b = a + 2; \\
S_3: &\quad c = 2; \\
S_4: &\quad d = b + c;
\end{aligned}
$$

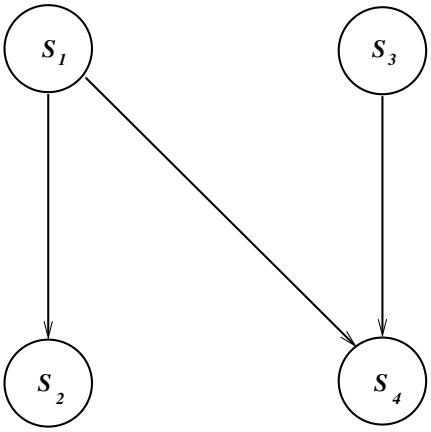
The dependence graph for the above example is shown below.



The variable b in statement S_4 can be replaced by a copy of the right hand side expression of statement S_2, the assignment to b.

$$
\begin{aligned}
S_1: &\quad a = 1; \\
S_2: &\quad b = a + 2; \\
S_3: &\quad c = 2; \\
S_4: &\quad d = (a + 2) + c;
\end{aligned}
$$

The modified dependence graph is



Statement substitution is illegal if it introduces spurious dependence arcs. In the statements shown below simple substitution of statement S_2 into statement S_4 would introduce the dependence $S_3 \delta S_4$ instead of $S_1 \delta S_4$.

$$
\begin{aligned}
S_1: &\quad a = 1; \\
S_2: &\quad b = a + 2; \\
S_3: &\quad a = 2; \\
S_4: &\quad d = b + c;
\end{aligned}
$$

Lemma 3.3.4.1 explains when simple statement substitution is legal. *Lemma* uses the concept of the dominator relationship. Statement S_μ dominates statement

S_ν if every path from the beginning of the program to S_ν goes through S_μ. Testing for the dominator relationship is explained in .

Lemma 3.3.4.1: If assignment statements S_μ and S_ν are in a loop L, and $S_\mu \delta S_\nu$ with dependence zero, then S_μ may be substituted into S_ν only if

i) S_μ dominates S_ν, and

ii) there is no statement S_ω between S_μ and S_ν with $S_\mu \delta^a S_\omega$ or $S_\mu \delta^o S_\omega$ with dependence distance zero.

Consider the loop shown below.

```
for (i = 0; i < N; i++)
{
    S1 :  a[i] = c[i] + b[i];
    S2 :  c[i] = e[i];
    S3 :  b[i+1] = a[i] + 2;
}
```

The above loop has a cycle $S_1 \delta S_3 \delta S_1$.

The dependence $S_1 \delta^f S_3$ with dependence distance zero.

Substitution of statement S_1 into S_3 is illegal since $S_1 \delta^a S_2$with dependence distance zero. However, reordering the statements allows statement substitution.

```
for (i = 0; i < N; i++)
{
        S1 :  a[i] = c[i] + b[i];
        S3 :  b[i + 1] = (c[i] + b[i]) + 2;
        S2 :  c[i] = e[i];
}
```

The above loop has a single statement cycle in S_3. Once the multistatement cycle has been reduced to a single statement cycle other vectorisation techniques such as loop distribution can be applied for vectorisation of the loop.

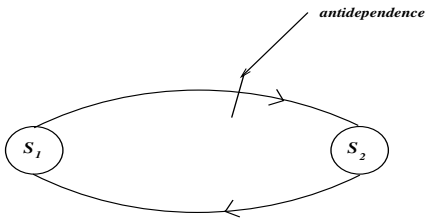
3.3.5 Index set splitting

When a cycle of data flow dependence and data anti dependence exists, index set splitting can be applied.

Consider the loop shown below.

```
for (i = 1; i <= 100; i+)
{
      S1 :  a[i] = b[101 − i] + c;
      S2 :  b[i] = e[i];
}
```

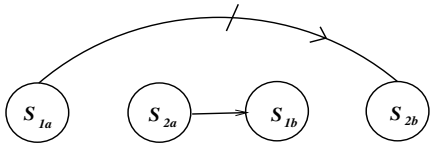
The dependence graph for the above loop is



By splitting the index set into disjoint parts, we break the dependence cycle.

```
for (i = 1; i <= 50; i++)
{
    S1a :  a[i] = b[101 - i] + c;
    S2a :  b[i] = e[i];
}
for (i = 51; i <= 100; i++)
{
    S1b :  a[i] = b[101 - i] + c;
    S2b :  b[i] = e[i];
}
```

The modified dependence graph is acyclic as shown below.



3.3.6 Loop fusion

Loop fusion merges two loops into a single loop. It was originally developed for vector machine compilers to recognise vectorisation in FORTRAN loops. In parallel computers, it is used to decrease the overhead of parallel loops.[Wolfe 1987]

The following disjoint parallel loops are identical in every respect except for the different names of the index variables.

```
for_all (i = 0; i < N; i++)
        d[i] = e[i] + f[i] + x[i];
for_all (j = 0; j < N; j++)
        e[j] = d[j] + f[j];
```

We get the following fused loop after applying loop fusion to the above loops.

```
for_all (i = 0; i < N; i++)
{
     d[i] = e[i] + f[i] + x[i];
     e[i] = d[i] + f[i];
}
```

3.3.7 Loop interchange

Vectorisation is performed in the inner loop rather than in the outer loop. Sometimes it is possible to perform loop interchange so as to enable vectorisation.

Consider the loop nest shown below.

$$\begin{aligned}
&\texttt{for}\,(i = 1; i <= N; i++)\\
&\quad \texttt{for}\,(j = 1; j <= N; j++)\\
&\qquad S_1: \quad a[i][j] = a[i][j-1] + a[i][j+1];
\end{aligned}$$

$S_1 \delta^f S_1$ with distance vector $[0 \;\; 1]^T$ and $S_1 \delta^a S_1$ with distance $[0 \;\; 1]^T$. Hence vectorisation of the inner loop is not possible. After interchanging, we get the loop nest shown below.

$$\begin{aligned}
&\texttt{for}\,(j = 1; j <= N; j++)\\
&\quad \texttt{for}\,(i = 1; i <= N; i++)\\
&\qquad a[i][j] = a[i][j-1] + a[i][j+1];
\end{aligned}$$

Now, the inner loop can be vectorised to get the code shown below.

$$\begin{aligned}
&\texttt{for}\,(j = 1; j <= N; j++)\\
&\qquad a[1:N][j] = a[1:N][j-1] + a[1:N][j+1];
\end{aligned}$$

For multiprocessor machines the outermost loop should be '`for_all`'. Thus we can move the innermost loop to outermost and get a '`for_all`' loop. Consider the loop nest shown below.

$$\begin{aligned}
&\texttt{for}\,(i = 1; i <= N; i++)\\
&\quad \texttt{for}\,(j = 1; j <= N; j++)\\
&\qquad a[i][j] = a[i-1][j] + a[i-2][j];
\end{aligned}$$

The outermost loop cannot be '`for_all`' since loop carried dependence exists in the outermost dimension. The dependence for the innermost dimension is zero and hence innermost loop can be converted into '`for_all`'. Interchanging the two loops we get the '`for_all` code shown below.

$$\begin{aligned}
&\texttt{for_all}\,(j = 1; j <= N; j++)\\
&\quad \texttt{for}\,(i = 1; i <= N; i++)\\
&\qquad a[i][j] = a[i-1][j] + a[i-2][j];
\end{aligned}$$

There are other goals for loop interchanging than considered above. For example, machines give better performance if loop limits are large. Vector machines give better performance with long vectors than short vectors. Multiprocessors perform better when many parallel iterations are possible. Target machine architecture also influences the use of loop interchanging for satisfying a particular goal.

Lemma 3.3.7.1: Legality of loop interchanging

The following conditions must be satisfied for loop interchanging to be legal.

i) the loops L_1 and L_2 must be tightly nested i.e. L_1 surrounds L_2 but contains no other executable statements.

ii) the loop limits of L_2 are not functions of index variable for L_1.

iii) there are no statements S_μ and S_ν (not necessarily distinct) in L_2 with a dependence relation $S_\mu \delta^* S_\nu$ with direction vector $[<\ >]^T$ where δ^* denotes any of flow, anti or output dependence.

3.4 Unimodular Transformations

Unimodular matrix plays an important role in the theory of loop transformations. It is a very convenient mathematical tool for expressing loop transformations.

Definition 3.3.7.1: Unimodular Matrix

A square integral matrix U is unimodular if $|U| = \pm 1$.

Thus inverse of U is i.e. U^{-1} exists and is unimodular.

Let $I \in Z^m$ be the index set of the original loop nest. Then the unimodular transformation of I is a mapping

$$T^u : I \rightarrow J \text{ such that } J \in Z^m \text{ and } J = UI$$

The mapping is one–to–one since if $\overline{i_1}, \overline{i_2} \in I$ and $U\overline{i_1} = U\overline{i_2}$ implies $\overline{i_1} = \overline{i_2}$.

The loop transformations such as interchange, reversal, skewing etc. can be expressed in terms of unimodular transformations. For example, a loop interchange transformation transforms iteration (i, j) to iteration (j, i). Hence loop interchange is a unimodular transformation with unimodular matrix $U = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ since $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} j \\ i \end{bmatrix}$.

Lemma: If $\overline{d}$ is the distance vector in the iteration space I then $U\overline{d}$ is the distance vector in the transformed iteration space J.

*Proof:*Let $\overline{i_1},\ \overline{i_2}\ \in I$ be the index points related by distance vector $\overline{d}$. Thus $\overline{i_2} - \overline{i_1} = \overline{d}$. Hence in transformed iteration space we get $U\overline{i_2} - U\overline{i_1} = U\overline{d}$. But $U\overline{i_2} = \overline{j_2} \in J$ and $U\overline{i_1} = \overline{j_1} \in J$. Hence we get $\overline{j_2} - \overline{j_1} = U\overline{d}$ i.e. dependence vector in transformed iteration space is $U\overline{d}$.

If $\overline{d} = [d_1\ \ d_2]^T$ is the dependence vector in loop interchange then after loop interchanging we get the transformed dependence vector $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} d_2 \\ d_1 \end{bmatrix}$.

3.4.1 Loop reversal

Reversal of the i^{th} loop in a loop nest is represented by the identity matrix with i^{th} diagonal element set to -1. Thus for two dimensions $U = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ represents the loop reversal matrix.

Consider the loop nest shown below.

$$\begin{aligned} &\texttt{for}\,(i = 1; i <= N; i++) \\ &\quad \texttt{for}\,(j = 1; j <= N; j++) \\ &\quad\quad a[i][j] = a[i-2][j+1]; \end{aligned}$$

After loop reversal we get the loop nest

$$\begin{aligned} &\texttt{for}\,(i = 1; i <= N; i++) \\ &\quad \texttt{for}\,(j = -1; j <= -N; j++) \\ &\quad\quad a[i][-j] = a[i-2][-j+1]; \end{aligned}$$

3.4.2 Loop skewing

Skewing loop i_j by an integer factor f with respect to loop i_i maps the iteration $(i_1, i_2, i_3, \ldots, i_{i-1}, i_i, i_{i+1}, \ldots, i_{j-1}, i_j, i_{j+1}, \ldots, i_n)$ to $(i_1, i_2, i_3, \ldots, i_{i-1}, i_i, i_{i+1}, \ldots, i_{j-1}, i_j + fi_i, i_{j+1}, \ldots, i_n)$.

The unimodular matrix U that produces skewing is the identity matrix with the element t_{ji} equal to f. Consider the loop nest shown below.

`for` $(i_1 = 0; i_1 <= 6; i_1 + +)$

`for` $(i_2 = 0; i_2 <= 8; i_2 + +)$

$a[i_1][i_2] = a[i_1][i_2 - 1] + a[i_1 - 1][i_2] + a[i_1 - 1][i_2 + 1]$;

The dependence matrix for this loop nest is $D = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & -1 \end{bmatrix}$. After loop skewing with $U = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$ we get the transformed loop nest as shown below.

`for` $(j_1 = 0; j_1 < 6; j_1 + +)$

`for` $(j_2 = j_1; j_2 <= j_1 + 8; j_2 + +)$

$a[j_1][j_2 - j_1] = a[j_1][j_2 - j_1 - 1] + a[j_1 - 1][j_2 - j_1] + a[j_1 - 1][j_2 - j_1 + 1];$

The dependence matrix D gets transformed to $D' = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$. Fig. shows the original index space and the transformed index space for the loop nest considered above.

3.4.3 Legality of Unimodular Transformations

A unimodular transformation of a loop nest must preserve the semantics of the program. If in the original loop nest index point $\overline{i_2}$ is executed after index point $\overline{i_1}$ where $(\overline{i_1}, \overline{i_2} \in I \; and \; \overline{i_2} = \overline{i_1} + \overline{d})$ then in the transformed index space $\overline{j_2}$ must

be executed after $\overline{j_1}$ to satisfy the dependence. Hence the condition for legality of unimodular transformation is $\forall \overline{d} \in D,\ U\overline{d} \succeq \overline{0}$.

3.4.4 Loop permutation

A permutation σ on a loop nest transforms iteration $(i_1, i_2, i_3, \ldots, i_n)$ to $(i_{\sigma_1}, i_{\sigma_2}, i_{\sigma_3}, \ldots, i_{\sigma_n})$. The transformation can be expressed in a matrix form as a unimodular transformation matrix U derived from the identity matrix I with rows permuted by σ . Loop interchange is a special case of loop permutation with $n = 2$.

Fully permutable loop nest

Consider a loop nest shown below

$$\begin{aligned}
&\texttt{for}\,(i = 1; i <= N; i++)\\
&\quad \texttt{for}\,(j = 1; j <= N; j++)\\
&\qquad a[i][j] = a[i-1][j-1] + a[i][j-1];
\end{aligned}$$

The dependence matrix for this loop nest is $D = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$. After permutation the dependence matrix gets transformed to $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$ i.e. $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$. Thus after permutation the dependencies remain lexicographically positive. Such loops for which each of the entries of the dependence matrix D is positive are called fully permutable loop nest.

Definition: Fully permutable loop nest

Loops I_i through I_j of a loop nest with dependence matrix D are fully permutable if and only if $\forall \overline{d} \in D : \Big((d_1, d_2, d_3, \ldots, d_{i-1}) \succ \overline{0}$ or $\forall i \leq k \leq j : d_k \geq 0\Big)$

Chapter 5

Tiling Transformation

Tiling is an important technique for extraction of parallelism. Informally, tiling consists of partitioning the iteration space into several chunk of computation called tiles (blocks) such that sequential traversal of the tiles covers the entire iteration space. Hence tiling increases the granularity of computation and decreases the amount of communication incurred between processors. This makes tiling very suitable for distributed memory architectures where communication startup costs are very high and hence frequent communication is undesirable. Since the iteration space of a tile is small compared to the entire iteration space, data locality and data reuse are higher.

Tiling of a nest of '`for`' loops is obtained by strip mining each of the '`for`' loop followed by a loop interchanging transformation. Strip mining divides each loop into strips and hence each loop becomes two loops., i.e.,

$$\texttt{for}\,(i_1 = 0; i_1 <= N; i_1 + +)$$

becomes

$$\texttt{for}\,(i_1 = 0; i_1 <= N; i_1 = i_1 + n\ \ )$$
$$\texttt{for}\,(i_1^{'} = i_1; i_1^{'} <= \min(i_1 + n - 1, N); i_1^{'} + +)$$

The above loop nest divides the original iteration space into strips each of size n. The outer loop acts as an index for each of the strip whereas the inner loop traverses the index points on a particular strip. Consider the nest of '`for`' loops shown below

$$\texttt{for}\,(i_1 = 0; i_1 <= N_1; i_1 + +)$$
$$\texttt{for}\,(i_2 = 0; i_2 <= N_2; i_2 + +)$$
$$a[i_1][i_2] = a[i_1 - 1][i_2] + a[i_1][i_2 - 1];$$

The dependence matrix for this loop nest is $D = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. After application of strip mining to each of the '`for`' loops we get the following code.

$$\texttt{for}\,(i_1 = 0; i_1 <= N_1; i_1 = i_1 + n_1)$$
$$\texttt{for}\,(i_1^{'} = i_1; i_1^{'} <= \min(i_1 + n_1 - 1, N_1); i_1^{'} + +)$$
$$\texttt{for}\,(i_2 = 0; i_2 <= N_2; i_2 = i_2 + n_2)$$
$$\texttt{for}\,(i_2^{'} = i_2; i_2^{'} <= \min(i_2 + n_2 - 1, N_2); i_2^{'} + +)$$
$$a[i_1^{'}][i_2^{'}] = a[i_1^{'} - 1][i_2^{'}] + a[i_1^{'}][i_2^{'} - 1];$$

Applying loop interchanging we get the tiled code as shown in Fig. 5.1.

$$
\begin{array}{l}
\texttt{for}\,(i_1 = 0; i_1 <= N_1; i_1 = i_1 + n_1) \\
\texttt{for}\,(i_2 = 0; i_2 <= N_2; i_2 = i_2 + n_2) \\
\texttt{for}\,(i_1^{'} = i_1; i_1^{'} <= \min(i_1 + n_1 - 1, N_1); i_1^{'} + +) \\
\texttt{for}\,(i_2^{'} = i_2; i_2^{'} <= \min(i_2 + n_2 - 1, N_2); i_2^{'} + +) \\
\quad a[i_1^{'}][i_2^{'}] = a[i_1^{'} - 1][i_2^{'}] + a[i_1^{'}][i_2^{'} - 1];
\end{array}
$$

Figure 5.1: **Tile code for the nest of for loops.**

The outer two **`for' loops run across the tiles whereas the inner two `for' loops execute the iterations within the tile. Thus tiling transformation maps an n–deep loop nest into a 2n–deep loop nest. Since tiling relies on loop interchanging, the nest of `for' loops must be fully permutable.**

The parameters n_1 and n_2 determine the size of tile. Varying n_1 and n_2 thus varies granularity of computation. Fig. 5.2 shows index set of the loop with loop bounds N_1 and N_2 and tile size $n_1 \times n_2$. It is clear from the figure that tiling satisfies the following conditions.

•

As mentioned earlier tiling increases data reuse. This fact is illustrated by applying tiling transformation to the matrix multiplication problem.

$$
\begin{array}{l}
\texttt{for}\,(i = 1; i <= N; i + +) \\
\quad \texttt{for}\,(j = 1; j <= N; j + +) \\
\quad\quad \texttt{for}\,(k = 1; k <= N; k + +) \\
\quad\quad c[i][j] = c[i][j] + a[i][j] * b[k][j];
\end{array}
$$

Consider reference to array such as b. For every iteration of i loop N^2 values of the array b are needed. If the cache line size is smaller than the size of array b,

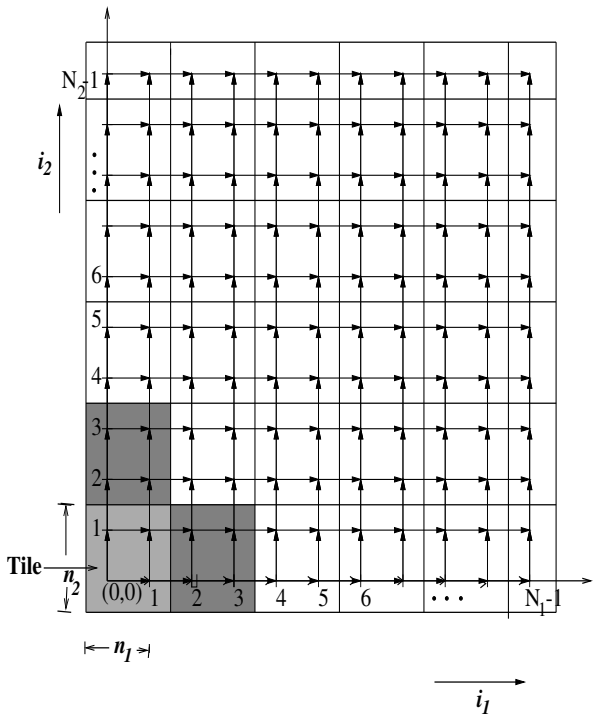


Figure 5.2: **Index set with tiling corresponding to tile code shown in 5.1**

the elements of array b may have to be reloaded for every iteration of the i loop. Consider tiled code corresponding to the above program as shown below.

$$
\begin{aligned}
&\texttt{for}\,(i = 1; i <= N; i = i + n)\\
&\texttt{for}\,(j = 1; j <= N; j = j + n)\\
&\texttt{for}\,(k = 1; k <= N; k = k + n)\\
&\texttt{for}\,(i^{'} = i; i^{'} <= \min(i + n - 1, N); i^{'} + +)\\
&\texttt{for}\,(j^{'} = j; j^{'} <= \min(j + n - 1, N); j^{'} + +)\\
&\texttt{for}\,(k^{'} = k; k^{'} <= \min(k + n - 1, N); k^{'} + +)\\
&\qquad c[i^{'}][j^{'}] = c[i^{'}][j^{'}] + a[i^{'}][j^{'}] * b[k^{'}][j^{'}];
\end{aligned}
$$

For every iteration of the $i^{'}$ loop n^2 values of the array b are needed. If n is chosen small enough to fit n^2 elements of the array in cache then reuse will result across the iterations of index $i^{'}$.

5.1 Tiling with Hyperplanes

Uptil now we have considered tiling with tiles that are rectangular in shape. However, in general, tiles can be of any arbitrary shape subject to certain constraints.We describe tiles by a set of hyperplanes. The equations of the hyperplanes determine the shape of the tile while the distance between the hyperplanes determine the size of the tile. Since all the tiles are identical in size and shape we concentrate on the fundamental tile, i.e., tile at the origin. Consider the fundamental tile as shown in Fig. 5.3

The tile can be represented by a $n \times n$ integer matrix P whose columns rep-

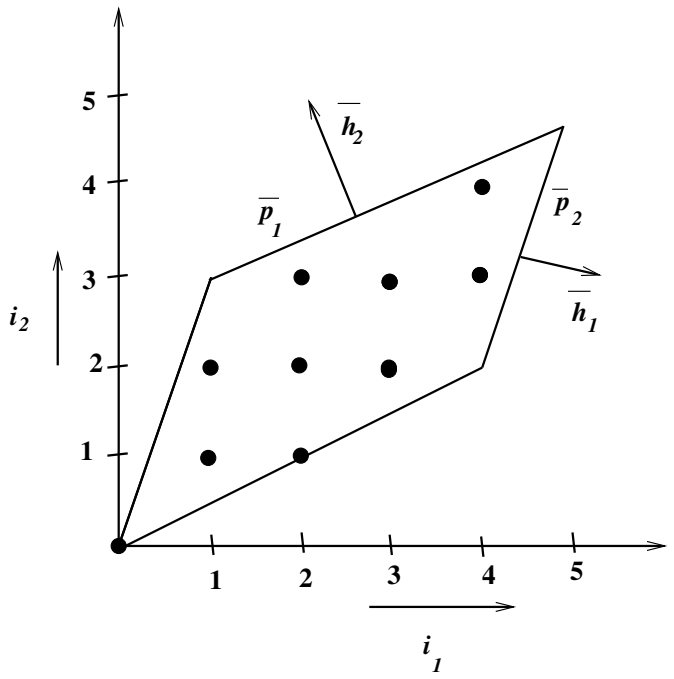


Figure 5.3: **Fundamental tile for tiling. $\overline{p_1}, \overline{p_2}$ are the vectors representing faces of the tile and $\overline{h_1}, \overline{h_2}$ are the vectors representing normal to faces of the tile**

resent the faces of the tile. Tiling can also be represented by a matrix $H \in Q^{n \times n}$ where each row of H, i.e., $h_i : i = 1, 2, 3, \ldots, n$ represents normal to face of the tile. Matrices P and H are related by $P = H^{-1}$. Index points belonging to the fundamental tile satisfy the condition $\overline{0} \leq H\overline{i} \leq \overline{1}$ For Fig. 5.3 $P = \begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix}$ and $H = \begin{bmatrix} \frac{3}{10} & \frac{-1}{10} \\ \frac{-2}{10} & \frac{4}{10} \end{bmatrix}$. The volume of the tile gives the number of index points within the tile. Let V_{calc} represent this volume. We get

$$V_{calc} = |P| = \frac{1}{|H|}$$

In Fig. 5.3 $V_{calc} = 10$.

Irigoin and Triolet imposed three constraints on H.

- H is non–singular, ensuring that the tile size is bounded, i.e., all the dimensions of the iteration space are tiled.
- $H^{-1} \in Z^{n \times n}$ is integral, ensuring that all tiles contain the same number of index points.
- $HD \geq 0$, ensuring that no cyclic dependencies exis between tiles. This is the atomic constraint. Each tile receives the data before start of computation. Once the data is received the computation completes without preemption and at the end results of computation are sent to the adjacent processor.

We define tiling as a transformation that transforms index space (Z^n) into tile space (Z^n) as show below

$$T : \overline{i} \rightarrow \overline{j} \text{ where } \overline{i} \in I,\ \overline{j} = [j_1\ j_2\ j_3\ \ldots j_n]^T = \left\lfloor H\overline{i} \right\rfloor$$

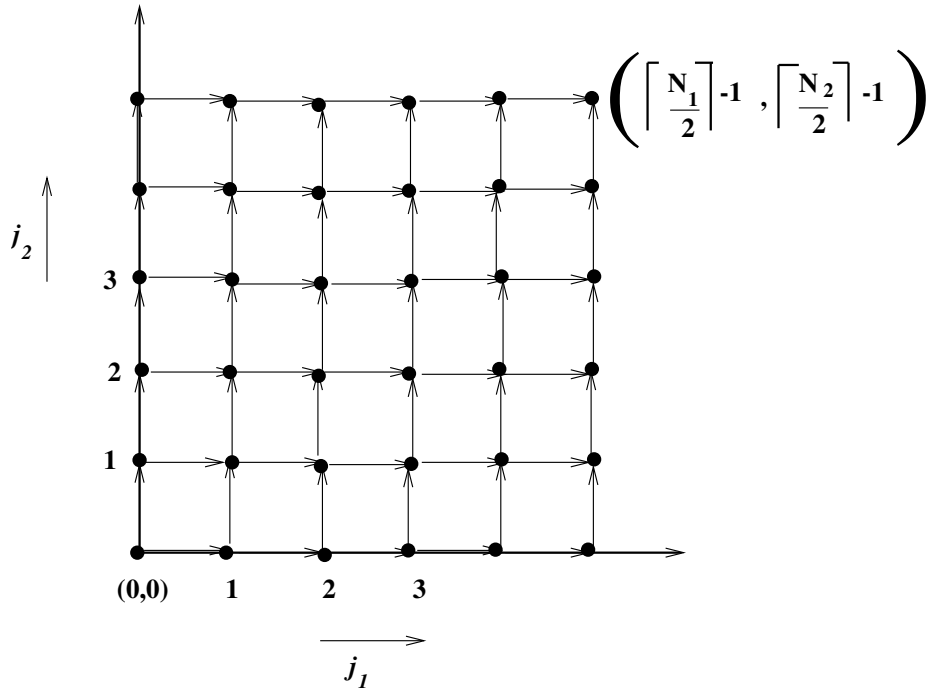


Figure 5.4: **Tile dependencies in the tile space corresponding to Fig. 5.1.**

Thus $\bar{i} = [i_1\ i_2\ i_3\ \ldots i_n]^T$ maps to $\left[\left\lfloor h_1\bar{i}\right\rfloor \quad \left\lfloor h_2\bar{i}\right\rfloor \quad \left\lfloor h_3\bar{i}\right\rfloor \ldots \left\lfloor h_n\bar{i}\right\rfloor\right]^T$ where $h_1, h_2, h_3, \ldots, h_n$ are rows of the matrix H as defined earlier. Consider the tiled code shown in Fig. 5.1. We get the tile space graph as shown in Fig.

Each node of the graph represents a tile. The dependencies in the index space get transformed into dependencies in the tile space. While mapping tiles on a processor array communication is required for satisfying these dependencies. Since it is desirable to have adjacent processor communication in a processor array, the tile dependencies must be of the form $d' = \left[d'_1\ d'_2\ d'_3 \ldots d'_n\right]$ where $d'_i \in \{0, 1\} : i = 1, \ldots n$.

Theorem 5.1 gives the condition on H for tile dependencies of this form.

Theorem 5.1: Let $\overline{d} \in D$ be the dependence vector in the index space I. Then

in the tile space $\overline{d}$ gets transformed to $d^{'} = \left[d_1^{'}\ d_2^{'}\ d_3^{'} \ldots d_n^{'}\right]$ where $d_i^{'} \in \{0, 1\}$: $i = 1, \ldots n$ if $H\overline{d} \leq \overline{1}$.

Proof: Let $\overline{i_1}, \overline{i_2} \in I$ be the two index points related by dependence vector $\overline{d}$,i.e., $\overline{i_2} = \overline{i_1} + \overline{d}$. Without loss of generality we can assume that $\overline{i_1}$ belongs to the fundamental tile. Let $\overline{i_2}$ belong to the adjacent tile. Then in the tile space we have

$$\left\lfloor H\overline{i_2} \right\rfloor = \left\lfloor H(\overline{i_1} + \overline{d}) \right\rfloor = \left\lfloor H\overline{i_1} + H\overline{d} \right\rfloor$$

Since $\overline{0} \leq H\overline{i_1} < \overline{1}$ for $\left\lfloor H\overline{i_2} \right\rfloor$ to belong to the neighbouring tile the condition $H\overline{d} \leq 1$ must be satisfied.

5.2 Scheduling and Execution of Tiles

We consider mapping of the tiles for executio on a processor array consisting of P processors. Each processor is connected to its neighbour o the right with which it communicates the data. All processors are identical and parameters of the communication link are identical.We also assume that a link exists between the last processor and the first processor.This communication link is required for cyclic block distribution as will be discussed later. Fig. shows the array of processors.All processors are identical and parameters of the communication link are identical.We also assume that a link exists between the last processor and the first processor.This communication link is required for cyclic block distribution as will be discussed later. Fig. 5.5 shows the array of processors.

As mentioned earlier each tile receives the data at the beginning of computation and sends the data at the end of computation. Hence synchronisation is

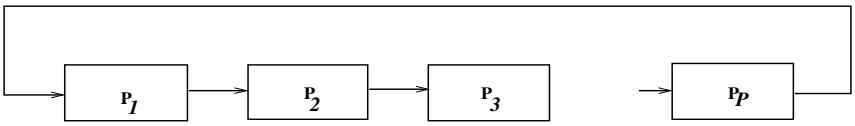


Figure 5.5: **Array of processors connected by communication links**

required only at the beginning and end of the executon of a tile. Each processor executes the tile code shown below.

```
receive();
compute();
send();
```

Fig. 5.6 shows execution of tiles corresponding to the tiled code shown in Fig. 5.1. Tiles are of rectangular shape and of size $n_1 \times n_2$. The index set is of size $N_1 \times N_2$. The dependence matrix is $D = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. In the first time step tile (0,0) is executed by processor P. At the end of execution it sends the computed values $a[1][0]$ and $a[1][1]$ to processor P_2. Each processor executes a column of tile. Hence no communication is required in the vertical direction since the processor remains same and data is available in main memory.Hence for the computed values $a[0][1], a[1][1]$ no communication is required for execution of tile (0,1). In the second time step processors P_1 and P_2 execute tiles (0,1) and (1,0) respectively. Similarly, in the third time step processors P_1, P_2 and P_3 execute tiles (0,2),(1,1) and (2,0) respectively. Thus we can imagine a wavefront sweeping across the tiles as shown in Fig. 5.6. At time step i all the tiles on i^{th} wavefront execute simultaneously. In our work we have used this wavefront approach for analysis of tiling.

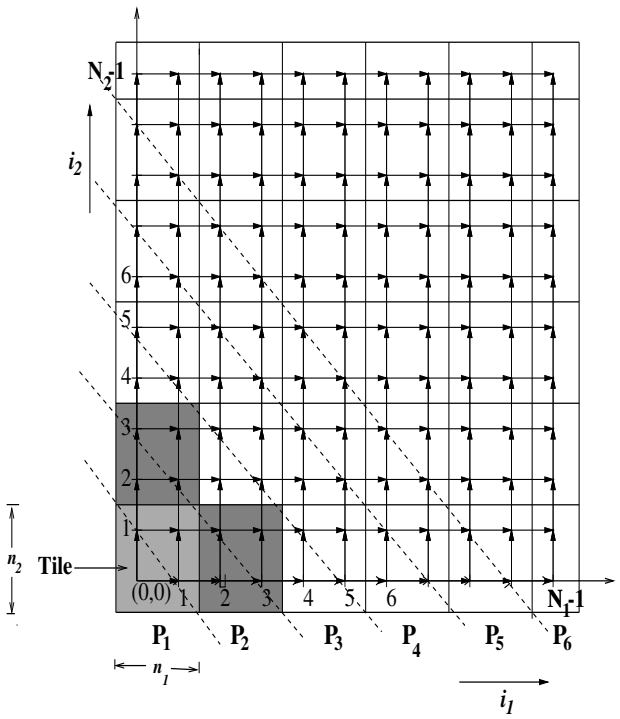


Figure 5.6: **Showing the index set and tiled code corresponding to Example 2.1. The dashed lines show the wavefront execution of the tiles.**