In general, for any stride $s$ and number of banks $M$, a stall will occur if

$$\frac{\text{LCM}(s, M)}{s} \leq \text{memory-access latency.}$$

Memory-bank conflicts will not occur if the stride is relatively prime to the number of banks.

If there are enough banks to prevent conflicts with a stride of 1, increasing the number of banks still further will decrease the stall frequency for some other strides.

For example, in the example above, if we had 64 memory banks instead of 16, stride-32 accesses would conflict how often?

How many banks would we need to prevent any conflicts with stride-32 accesses?

In the 1990s, most vector supercomputers have at least 64 banks, and some have as many as 512.

**A simplified model for vector performance**

[H&P §B.3] The execution time of a strip-mined loop is composed of three parts.

1. *The per-loop overhead.* This overhead comes from
   - computing the start addresses for each vector and
   - setting up the strip-mining loop control

   This overhead occurs once for the entire vector operation. We will call it $T_{\text{base}}$.

2. *The per-iteration overhead.* This overhead is incurred for each strip-mined block of vector instructions. It consists of—
   - the cost of executing the scalar code to control strip-mining of each strip, $T_{\text{loop}}$, plus
   - the vector start-up cost for each convoy, $T_{\text{start}}$. 
3. **The per-convoy time.** This is the time to execute the vector operations in the loop on a single convoy, $T_{chime}$.

These components make up the running time for a sequence of instructions on a vector of length $n$, according to the following formula:

$$ T_n = T_{base} + \left\lfloor \frac{n}{MVL} \right\rfloor (T_{loop} + T_{start}) + nT_{chime} $$

For simplicity’s sake, we will assume that $T_{start}$ and $T_{loop}$ are constant. Based on Cray-1 measurements, we can choose

$$ T_{base} = 10 \text{and} T_{loop} = 15. $$

Although each of these times is composed of the execution of several instructions, the values are not larger, because the scalar instructions that cause the overhead can be overlapped with the vector instructions.

What are some of the instructions that make up these overheads?

**Example:** Let us consider the time for the vector operation

$$ A = B \times s $$

where $s$ is a scalar and the length of the vectors $A$ and $B$ is 200.

Assume— the address of $A$ is initially in RA,
the address of $B$ is initially in RB,
$s$ is in FS, and
R0 permanently contains the value 0.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>ADDI</td>
<td>R4,R0,#1600</td>
<td>Number of bytes in vector.</td>
</tr>
<tr>
<td>2.</td>
<td>ADD</td>
<td>R2,R4,RA</td>
<td>Find end of $A$ vector.</td>
</tr>
<tr>
<td>3.</td>
<td>ADDI</td>
<td>R1,R0,#8</td>
<td>Length of first portion of vec.</td>
</tr>
<tr>
<td>4.</td>
<td>MOVI2S</td>
<td>VLR,R1</td>
<td>Load vector length</td>
</tr>
<tr>
<td>5.</td>
<td>ADDI</td>
<td>R1,R0,#64</td>
<td>Length <em>in bytes</em> of 1st portion</td>
</tr>
<tr>
<td>6.</td>
<td>ADDI</td>
<td>R3,R0,#64</td>
<td>Length of remaining portions</td>
</tr>
</tbody>
</table>
loop:
7. LV V1,RB Load B.
8. MULTVS V2,V1,FS Vector \infty scalar
9. SV RA,V2 Store A.
10. ADD RA,RA,R1 Start of next portion of A.
11. ADD RB,RB,R1 Start of next portion of B.
12. ADDI R1,R0,#512 Length in bytes of next portion
13. MOVI2S VLR,R3 Set vector length to 64.
14. SUB R4,R2,RA At the end of A?
15. BNZ R4,loop If not, go back.

We have assumed that \( T_{base} = 10 \) and \( T_{loop} = 15 \).

Because we have a load, multiply, and store for each vector element, \( T_{chime} = 3 \).

That is everything we need to know except \( T_{start} \). What makes up \( T_{start} \)?

- a vector-load startup of 12 clock cycles;
- a _____-cycle startup penalty for the multiply, and
- a _____-cycle startup for the store.

Thus, \( T_{start} = \)

Now we can plug all the numbers into the formula.

\[
T_n = T_{base} + \left\lceil \frac{n}{MVL} \right\rceil (T_{loop} + T_{start}) + nT_{chime}
\]

\[
T_{200} =
\]

\[
=
\]

\[
=
\]

What is the execution time per element?

This compares with an ideal execution time per element of 3.
Here is a graph of how overhead varies for vectors of different sizes. The vector startup cost amounts to about half the total overhead per element.

Notice that—

- For short vectors, the total overhead is more than half of the time per element. For long vectors, it reduces to about \( \frac{1}{2} \) of the total time.
- Why do the sudden jumps occur in the curve?

How much do these operations increase \( T_n \) by?

**Efficient memory access**

As we have seen, if arrays are stored in memory in the straightforward row-major way, efficient access to a row is possible, but not to a column:

\[
\begin{array}{cccc}
: & : & : & :
\end{array}
\]

Or, we can store in *column-major* format for efficient access to columns but not to rows:

\[
\begin{array}{cccc}
: & : & : & :
\end{array}
\]

Suppose
- \( d \) is the memory-access delay, and
- \( M \) is the number of memory modules.
If the memory is accessed in the “efficient access” direction, it can deliver one operand per clock cycle if (what condition involving $d$ and $M$ holds?)—

If this condition does not hold, accesses to what length vectors $N$ result in memory conflicts?

Thus, the aggregate memory bandwidth must be great enough to support aggregate memory demand.

If an instruction requires two input and one output stream, the aggregate bandwidth must be at least 3 items per clock period.

How many memory modules are needed to achieve this?

To achieve efficient access, we can

- alter the structure of the memory, or
- alter the structure of the data in memory.

Let us examine one way of achieving efficient access, by altering the structure of the memory, as was first done in the BSP.

*The Burroughs Scientific Processor (BSP):* A successor to the Illiac IV. Was intended to be a commercial product, but didn’t make it. The project was suspended in 1979.

A major innovation was the organization of memory to allow simultaneous access to rows, columns, and diagonals of arrays.

Consider, for example, a 4-column array laid out in a 4-way interleaved memory.

```
<table>
<thead>
<tr>
<th>A[0, 0]</th>
<th>A[0, 1]</th>
<th>A[0, 2]</th>
<th>A[0, 3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

A row of this array can be accessed efficiently by the processors, but not a column.
The BSP had 16 arithmetic units (AEs). All of them could attempt an access in parallel.

If the BSP had 16 memory modules, what would happen?

Instead the BSP had 17 memory modules.

Then, for a given memory address \( a \), which module does it fall in?

What is the formula for the offset within the module of memory address \( a \)?

In the BSP, in what width arrays will conflicts occur for column accesses?

In what width arrays will conflicts occur for \textit{diagonal accesses}?

\begin{itemize}
  \item What is the inter-element \textit{stride} of diagonal elements?
  \item For what width arrays will the stride be a multiple of the number of modules?
\end{itemize}

(Actually, the BSP compilers stored arrays in \textit{column-major} order, so to be perfectly accurate, “row” and “column” should be interchanged in the discussion above.)

Advantages of this scheme:

\begin{itemize}
  \item Very few array operations will experience memory conflicts, since—
    \begin{itemize}
      \item multiples of 17 are unusual array widths, and
      \item 17 is a prime number.
    \end{itemize}
  \item Little unused memory bandwidth. Why?
\end{itemize}
Restructuring data in memory

Instead of doing this (to achieve efficient access), an alternative would be to alter the structure of the data in memory.

For example, we could arrange an $8 \times 8$ array to skip every ninth memory word:

<table>
<thead>
<tr>
<th>(0,0)</th>
<th>(0,1)</th>
<th>(0,2)</th>
<th>(0,3)</th>
<th>(0,4)</th>
<th>(0,5)</th>
<th>(0,6)</th>
<th>(0,7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,0)</td>
<td>(1,1)</td>
<td>(1,2)</td>
<td>(1,3)</td>
<td>(1,4)</td>
<td>(1,5)</td>
<td>(1,6)</td>
<td>(1,7)</td>
</tr>
<tr>
<td>(2,0)</td>
<td>(2,1)</td>
<td>(2,2)</td>
<td>(2,3)</td>
<td>(2,4)</td>
<td>(2,5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2,6)</td>
<td>(3,0)</td>
<td>(3,1)</td>
<td>(3,2)</td>
<td>(3,3)</td>
<td>(3,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3,6)</td>
<td>(3,7)</td>
<td>(4,0)</td>
<td>(4,1)</td>
<td>(4,2)</td>
<td>(4,3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4,4)</td>
<td>(4,5)</td>
<td>(4,6)</td>
<td>(4,7)</td>
<td>(5,0)</td>
<td>(5,1)</td>
<td>(5,2)</td>
<td></td>
</tr>
<tr>
<td>(5,3)</td>
<td>(5,4)</td>
<td>(5,5)</td>
<td>(5,6)</td>
<td>(5,7)</td>
<td>(6,0)</td>
<td>(6,1)</td>
<td></td>
</tr>
<tr>
<td>(6,2)</td>
<td>(6,3)</td>
<td>(6,4)</td>
<td>(6,5)</td>
<td>(6,6)</td>
<td>(6,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(7,1)</td>
<td>(7,2)</td>
<td>(7,3)</td>
<td>(7,4)</td>
<td>(7,5)</td>
<td>(7,6)</td>
<td>(7,7)</td>
<td></td>
</tr>
</tbody>
</table>

Successive column elements, as well as successive row elements, lie in different modules.

How many quantities do we have to give to specify a vector operand?

1. Starting address.
2. Number of elements.
3. Precision (number of bits/element).
4.

If the stride is relatively prime to $M$, then $M$ successive memory accesses for that stride are directed to $M$ distinct memories.

In general, for any stride $s$, $M$ successive accesses of stride $s$ are directed to $\frac{M}{\text{GCD}(s, M)}$ different memories.

Since $M$ is usually a power of 2, what kind of strides give efficient accesses?

Hence, we can always add a “wasted” column to an array to provide a storage structure that can support efficient row and column accesses.

However, other kinds of accesses are also important:
• matrix diagonals in the major and minor directions, and
• square subarrays.

How is the stride of a diagonal access related to the stride of a column access?

So we can’t have efficient column and diagonal accesses if the number of memory modules is a power of 2.

In the BSP, of course, the number of modules was not a power of 2.

• Memory was not pipelined.
• Rather, in one memory cycle, the memory system delivered a block of 17 words, each from a separate memory.
• The input alignment network shrank a 17-way access to 16 operands by deleting some operand and compressing the remaining operands into a contiguous 16-element vector.
• The output alignment network did the reverse.
**Simple example: a 5-module memory**

A $4 \times 4$ matrix has two dummy columns stored, so it was stored as a $4 \times 6$ matrix.

- Rows are stored with a stride of 1.
- Columns are stored with a stride of 6
- Diagonals are stored with a stride of 7.

All are relatively prime to 5.

The diagram at the right shows how it is possible to access columns (top diagram) and diagonals (bottom diagram) efficiently.

However, what matrix access that we have mentioned is inefficient in this organization?

Another organization permits simultaneous access to even minor diagonal elements and square subarrays:
These elements are simultaneously accessible:

- rows
- columns (\(\Delta\))
- diagonals (o)
- square subarrays (□).

The squares cover the elements of the lower left half subarray.

\[
\begin{array}{cccc}
  a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\
  a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\
  a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\
  a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \\
\end{array}
\]

Making use of this clever method of storage required

- Sophisticated ways of removing the \(N\) desired items from the \(N+1\) items delivered by the \(N+1\) memories, and routing them to \(N\) processors.
- In other words, a sophisticated way of deciding which of the \(N\) memories to access.

This had its costs.

- Addressing was more complex.
- The matrix needs to be held in main memory. So memory accesses will be frequent.
Prime-number memory systems

Recently\(^1\) it has been noticed that the Chinese Remainder Theorem provides an easy way to map memory addresses to memories that are \(p\)-way interleaved, where \(p\) is a prime number.

Note that the mapping of the word at address \(a\) to memory is defined by two functions:

- \(m(a)\), the number of the memory bank that address \(a\) is mapped to.
- \(l(a)\), the local address within memory bank \(m(a)\).

Most current computers use the mapping

\[
\begin{align*}
m(a) &= a \mod N \\
l(a) &= a / N
\end{align*}
\]

(This is just low-order interleaving.)

The most desirable number of modules is prime, because then, only strides that are multiples of the number of modules will cause conflicts.

- Computing the memory bank number for a word at address \(a\) requires the computation of \(a \mod N\). It is possible to implement very fast hardware to compute a modulus.

- Computation of the local displacement in the memory bank requires a Euclidean division by \(N\). This division is quite complex when \(N\) is an odd number.

However, the Chinese Remainder Theorem can be used to define a faster mapping for a \(p\)-way interleaved memory.

The Chinese Remainder Theorem states that—

If \((m_i, m_j)\) are relatively prime for all integers \(i\) and \(j\) such that \(0 < i, j \leq n\), then the system

\[ x + c_1 \pmod{m_1}, \]
\[ x + c_2 \pmod{m_2}, \]
\[ \vdots \]
\[ x + c_n \pmod{m_n}, \]

has a unique solution modulo \( m \) where \( m = m_1 m_2 \ldots m_n \).

**Example:** Find an integer \( x \) such that—
\[ x + 2 \pmod{5} \]
\[ x + 1 \pmod{3} \]

If \( x + 2 \pmod{5} \), then \( x = 2 + 5k \), where \( k \) is an integer. By substitution in the second linear congruence,
\[ 2 + 5k + 1 \pmod{3} \]
\[ 2k + -1 \pmod{3} + 2 \pmod{3} \]
\[ k + 1 \pmod{3} \]

Hence, \( x = \cdots \pmod{3 \cdot 5} \).

How can we use the Chinese Remainder Theorem to compute a fast mapping for address \( a \)? Notice that if the number of modules is a prime \( p \), what we really need to compute is the address \( a \) modulo two relatively prime numbers. Which two?

\[ m(a) = a \pmod{p} \]
\[ l(a) = a \pmod{ } \]

Fast hardware can perform both of these computations. This provides a way to use a prime number of memory modules directly, without leaving any words unused.
Hierarchical memories
Consider the Cray I architecture. It used hierarchical memories.

Main memory is separated from the processing units by one or two levels of intermediate memories.

- For vector operations, there are
  - eight $V$ registers; each can hold 64 double-precision numbers.

- For scalar operations, there are
  - eight 64-bit $S$ registers (the fastest level), and
  - sixty-four 64-bit $T$ registers (slower but still high-speed).

Data that cannot be held (“cached”) in the $S$ registers overflows into the $T$ registers.

- For addresses, there are
  - eight 24-bit $A$ registers (fastest),
  - sixty-four 24-bit $B$ registers (slower).

The difference between these registers and ordinary caches is that these registers are programmable. They must be explicitly loaded and stored.

There is another non-programmable cache: the instruction buffer.
Because it holds tight loops, instruction fetches tend to be rare events.

The Cray I design allows operands to be fetched mostly from high-speed registers, where one operand can be provided per cycle, regardless of the access pattern.

Intermediate results can be held in registers, so main memory can be slower.

Whether this organization, or the optimized memory organization, is cheaper depends on technology.

However, the performance of programmable registers can benefit from advances in software as well as hardware.

Moreover, if a multiprocessor uses registers instead of a cache, there is no need for a cache-coherence protocol.