A supercomputer is a hardware and software system that provides close to the maximum performance that can currently be achieved.

Over the last 10 years the range for the Top500 (http://www.top500.org) has increased greater than Moore’s Law:

- 1993:
  - #1 = 59.7 GFlop/s
  - #500 = 422 MFlop/s
- 2004:
  - #1 = 70 TFlop/s
  - #500 = 850 GFlop/s
Emergence of supercomputers

Source: J. Dongarra
HPC Challenge Benchmarking
http://icl.cs.utk.edu/hpcc/
http://icl.cs.utk.edu/hpcc/hpcc_results_all.cgi
Basic Idea
- **Extend the instruction set and architecture of the system to support the execution of commonly used vector operations in hardware.**
- **Vector Operations** work on vectors, linear arrays of numbers.
  - Typical vector operation might add two 64-element, floating-point vectors to obtain a single 64-element vector results.

Example:
- $A[i] = B[i] + C[i], \ i = 1, 64$
  - are represented by a vector instruction of the type
    
    $$\text{VOP V3, V1, V2}$$

  - where VOP represents a vector operation, and V1, V2, V3 indicate specific vector registers.

  - The operation performed is:
    $$\text{V3 = V1 VOP V2}$$
  - for all register values within each vector registers.

\[1/11/2005 \text{ Hiroaki Kobayashi} \]
Vector registers

- Each vector register is a fixed-length bank holding a single vector and must have at least two read ports and one write port.

Vector functional units

- Each unit is fully pipelined and can start a new operation on every clock cycle. A control unit is needed to detect hazards, both from conflicts for the functional unit and from conflicts for register accesses.

Vector load-store unit

- This is a vector memory unit that loads or stores a vector to or from memory. Vector loads and stores are fully pipelined, so that words can be moved between the vector registers and memory with a bandwidth of 1 word per clock cycle, after an initial latency.

### Remarks

1. The clock rates shown are for the vector units.
2. The lanes show the number of parallel pipelines used to execute operations within each vector instruction.
3. The Fujitsu machines’ vector registers are configurable.
4. The NEC machines have eight foreground vector registers connected to the arithmetic units plus 32-64 background vector registers connected between the memory system and the foreground vector registers.
5. The Cray SV1 can group four CPUs with 2 lanes each to act in unison as a single larger CPU with eight lanes, called a multi-streaming Processor (MSP).
SAXPY (Single-precision a \( \times \) X plus Y) and DAXPY (Double-precision a \( \times \) X plus Y)

\[ Y = a \times X + Y, \quad \text{where } X \text{ and } Y \text{ are vectors, initially resident in memory, and } a \text{ is a scalar} \]

Many scientific applications include this type of problems.

How is SAXPY/DAXPY translated into scalar and vector codes?

(Assume that the number of elements, or length, of a vector register (64) matches the length of the vector operation)

<table>
<thead>
<tr>
<th>MIPS code</th>
<th>Vector code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD ( F_0 ), ( a ) ; load scalar ( a )</td>
<td>LD ( F_0 ), ( a ) ; load scalar ( a )</td>
</tr>
<tr>
<td>DADDIU ( R_4 ), ( R_x ), #512 ; last address to load</td>
<td>DADDIU ( R_4 ), ( R_x ), #512 ; last address to load</td>
</tr>
<tr>
<td>LD ( F_2 ), ( 0(\text{R}_x) ) ; load ( X(i) )</td>
<td>MUL.V D ( V_2 ), ( V_1 ), ( F_0 ) ; vector-scalar multiply</td>
</tr>
<tr>
<td>MUL.D ( F_2 ), ( F_0 ) ; ( \times ) ( X(i) )</td>
<td>LD ( V_3 ), ( \text{R}_y ) ; load vector ( Y )</td>
</tr>
<tr>
<td>LD ( F_4 ), ( 0(\text{R}_y) ) ; load ( Y(i) )</td>
<td>ADDV.D ( V_4 ), ( V_2 ), ( V_3 ) ; add</td>
</tr>
<tr>
<td>ADD.D ( F_4, F_2, F_4 ) ; ( a \times X(i) + Y(i) )</td>
<td>SV ( \text{R}_y, V_4 ) ; store the result</td>
</tr>
<tr>
<td>SD ( 0(\text{R}_y), F_4 ) ; store into ( Y(i) )</td>
<td></td>
</tr>
<tr>
<td>DADDIU ( R_x ), ( R_x ), #8 ; increment index ( X )</td>
<td></td>
</tr>
<tr>
<td>DADDIU ( R_y ), ( R_y ), #8 ; increment index ( Y )</td>
<td></td>
</tr>
<tr>
<td>DSUBU ( R_20 ), ( R_4 ), ( R_x ) ; compute bound</td>
<td></td>
</tr>
<tr>
<td>BNEZ ( R_20 ), Loop ; check if done</td>
<td></td>
</tr>
</tbody>
</table>

The vector processor greatly reduces the dynamic instruction bandwidth. Executing only 6 instructions versus almost 600 for MIPS.

This reduction occurs both
1. because the vector operations work on 64 elements and
2. because the overhead instructions that constitute nearly half the loop on MIPS are not present in the vector code.
MIPS code

LD F0,a ; load scalar a
DADDIU R4,Rx,#512 ; last address to load
Loop LD F2,0(Rx) ; load X(i)
MUL.D F2,F2,F0 ; a \times X(i)
ADD.D F4,0(Ry) ; load Y(i)
ADDV.D V4,V2,V3 ; add
SD 0(Ry),F4 ; store into Y(i)
DADDIU Rx,Rx,#8; ; increment index to X
DADDIU Ry,Ry,#8; ; increment index to Y
DSUBU R20,R4,Rx ; compute bound
BNEZ R20, Loop ; check if done

Vector code

LD F0,a ; load scalar a
LV V1,Rx ; load vector X
MULV.D V2,V1,F0 ; vector-scalar multiply
LV V3,Ry ; load vector Y
ADDV.D V4,V2,V3 ; add
SV Ry,V4 ; store the result

A large decrease in the frequency of pipeline interlock in vector processing

In the straightforward MIPS code, every ADD.D must wait for a MUL.D and every S.D must wait for the ADD.D.
On the vector processor, each vector instruction will only stall for the first element in each vector, and then subsequent elements will flow smoothly down the pipeline.

Pipeline stalls: only once per vector operation vs. once per vector element!

Three factors that affect the execution time of a sequence of vector operations:
- Length of the operand vectors
- Structural hazards among the operations
- Data dependency

The time for a single vector instruction can be computed when the vector length and the initiation rate are given.

Initiation rate: the rate at which a vector unit consumes new operands, and produces new results.

Model for estimating vector execution time:
- One lane with an initiation rate of one element per clock cycle for individual operations

Notion of a convoy and a chime:
- A convoy is a set of vector instructions that could potentially begin execution together in one clock period.
- The instructions in a convoy must not contain any structural hazards or data hazards.
- A timing metric, called chime, can be used for estimating the performance of a vector sequence consisting of convoys.
- A chime is the unit of time taken to execute one convoy.
• The sequence requires four convoys and hence takes four chimes.

- Ignores some overheads: vector instruction issue overhead and the vector start-up time.

Because the initiation overhead is very small compared with cycles for a vector operation, the chime approximation is reasonably accurate for long vectors.

For example, for 64-element vectors, the sequence would take about 256 clock cycles but the initiation overhead is as small as two cycles in most vector machines.

Space-time diagram

- Depict the processing status of a pipeline as a function of time

- Start-up time is the pipeline latency of the vector operation, and is principally determined by how deep the pipeline is for the functional unit used.

A deeper pipeline has a longer start-up time!
Question:
If the startup overhead is assumed as shown in the table, Show the time that each convoy can begin and total number of cycles needed, and how does the time compare to the chime approximation for a vector length of length 64?

<table>
<thead>
<tr>
<th>Unit</th>
<th>Start-up overhead (cycles)</th>
<th>Convoy</th>
<th>Starting time</th>
<th>First-result time</th>
<th>Last-result time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load and store unit</td>
<td>12</td>
<td>1. LV</td>
<td>0</td>
<td>12</td>
<td>12 + n</td>
</tr>
<tr>
<td>Multiply unit</td>
<td>7</td>
<td>2. MULVS.D</td>
<td>12 + n</td>
<td>12 + n + 12</td>
<td>23 + 2n</td>
</tr>
<tr>
<td>Add unit</td>
<td>6</td>
<td>3. ADDV.D</td>
<td>24 + 2n</td>
<td>24 + 2n + 6</td>
<td>28 + 3n</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. SV</td>
<td>30 + 3n</td>
<td>30 + 3n + 12</td>
<td>41 + 4n</td>
</tr>
</tbody>
</table>

Time per result for a vector of length 64 is $4\times(41/64)=4.64$ clock cycles

Start-up time add a 16\% overhead to the execution time!

- The start-up time for a load is the time to get the first word from memory into a register.
  - If the rest of the vector can be supplied without stalling, then the vector initiation rate is equal to the rate at which new words are fetched or stored.
  - Unlike simpler functional units, the initiation rate may not necessarily be 1 clock because memory bank stalls can reduce effective throughput.

Startup-penalties of Cray 1

<table>
<thead>
<tr>
<th>Operation</th>
<th>Start-up penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector add</td>
<td>6</td>
</tr>
<tr>
<td>Vector multiply</td>
<td>7</td>
</tr>
<tr>
<td>Vector divide</td>
<td>20</td>
</tr>
<tr>
<td>Vector load</td>
<td>12</td>
</tr>
</tbody>
</table>

To maintain an initiation rate of 1 word fetched or stored per clock, the memory system must be capable of producing or accepting this much data.

Banked Memory System
Three primary reasons for use of memory banks for vector processing, rather than simple interleaving:

1. The memory system needs to have multiple banks and able to control the addresses to the banks independently.
   - Many vector computer support multiple loads or stores per clock, and the memory bank cycle time is often several times larger than the CPU cycle time.

2. Many vector processors support the ability to load or store data words that are not sequential.
   - Independent bank addressing, rather than interleaving, is required.

3. Many vector computers support multiple processors sharing the same memory system, and so each processor will be generating its own independent stream of address

Cray T90
- 2.167 ns CPU clock cycle
- 32 processors, each capable of four loads and two stores per CPU clock cycle
- SRAM memory cycle of 15ns

Calculate the minimum number of memory banks required to allow all CPU to run at full memory bandwidth

- The maximum number of memory references each cycle is 192 (=32CPUs x 6 references/CPU)
- Each SRAM bank is busy for 15/2.167 = 6.92 clock cycles, rounding up 7 clock cycles.

192 \( \div 7 \) = 1344 memory Banks!

But actually, 1024 memory banks are adopted (later model introduce synchronous DRAM for providing sufficient bandwidth).
Suppose we want to fetch a vector of 64 elements starting at byte address 136, and a memory access takes 6 clocks.

Questions:
• How many memory banks must we have to support one fetch per clock cycle?
• With what addresses are the banks accessed?
• When will the various elements arrive at the CPU?

Vector-Length Control

- A single piece of code may require different vector lengths
  - What do you do when the vector length in a program is not exactly 64, the physical size of a vector register?

Example:

```plaintext
do 10 I = 1,n
  10 Y(i) = a * X(i) + Y(i)
```

The size of all the vector operations depends on `n`, which may not even be known until runtime! The value of `n` might also be a parameter to a procedure containing the above loop and therefore be subject to change during execution.

Solution

```
Vector-Length Register
```

The introduction of the VLR solves the vector-length problem as long as the real length is less than or equal to the maximum vector length (MVL) defined by the processor.
Strip mining

- Generation of code such that each vector operation is done for a size less than or equal to the MVL.

\[
\text{do 10 I = 1,n} \\
10 \quad Y(i) = a \times X(i) + Y(i)
\]

The inner loop of the code is vectorizable with length VL.

There are two key factors that contribute to the running time of a strip-mined loop consisting of a sequence of convoys:

1. The number of convoys in the loop, which determines the number of chimes.
   - \( T_{\text{chime}} \) for the execution time in chimes
2. The overhead for each strip-mined sequence of convoys.
   - This overhead consists of the cost of executing the scalar code for strip-mining each block, \( T_{\text{loop}} \), plus the vector start-up cost for each convoy, \( T_{\text{start}} \)

Total running time for a vector sequence operating on a vector of length \( n \), called \( T_n \):

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

\( T_{\text{start}}, T_{\text{loop}}, \text{and } T_{\text{chime}} \) are compiler and processor dependent.

The register allocation and scheduling of the instructions affect both what goes in a convoy and the start-up overhead of each convoy.
What is the execution time for the vector operation 
\( A = B \times s \), where \( s \) is a scalar and the length of the vector \( A \) and \( B \) is 200?

Answer
- Since \((200 \text{ mod } 64)=8\), the first iteration of the strip-mind loop will execute for a vector length of 8 elements, and the following iterations will execute for a vector length of 64 elements.
- Since the vector length is either 8 or 64, increment the address registers by \(8 \times 8 = 64\) after the first segment and \(8 \times 64 = 512\) for later segments.
- Since the total number of bytes in the vector is \(8 \times 200 = 1600\), the test for completion is performed by comparing the address of the next vector segment to the initial address plus 1600.

The addresses of \( A \) and \( B \) are initially in \( Ra \) and \( Rb \), \( s \) is in \( Fs \), and \( R0 \) always holds 0.
- \( T_{\text{loop}} \) is a constant value of 15 from Cray-1 data.

\[ \text{Answer: } 784 / 200 = 3.9 \text{ compared with a chime approximation of } 3. \]
A chime counting model would lead to 3 clock cycles per element, while the two sources of overhead add 0.9 clock cycles per element in the limit!

- The second problem addresses is that the position in memory of adjacent elements in a vector may not be sequential.

Example:
```fortran
  do 10 i=1, 100
    do 10 j=1,100
      A(i,j)=0.0
    do 10 k=1,100
  10    A(i,j)=A(i,j)+B(i,k)*C(k,j)
```

At the statement labeled 10 we could vectorize the multiplication of each row of B with each column of C and strip-mine the inner loop with k as index variable.

To do so, we must consider how adjacent elements in B and adjacent elements in C are addressed.

- If the elements of the vectors are stored in the column-major order (the case in which the code is written in Fortran), the elements of B accessed by iterations in the inner loop are separated by the row size times 8 (the number of bytes per entry) for a total of 800 bytes.

The distance separating elements are gathered into a single register, and processed as if it had logically adjacent elements.
Suppose
- 8 memory banks
- A total memory latency of 12 cycles
  - The time from when the address arrives at the bank until the bank returns a data value. The latency includes the start-up cost of fetching a vector from memory.
- A bank busy time of 6 cycles
  - The time the bank is occupied with one request. A processor cannot issue a second request to the same bank until the bank busy time has elapsed
- How long will it take to complete a 64-element vector load with a stride of 1?
- With a stride of 32?

Case of 1-Stride:
- Since the number of banks is larger than the bank busy time, for a stride of 1, no conflict on memory banks
  - The load will take 76 (=12+64) clock cycles, 1.2 clocks per element

Case of 32-Stride:
- Every access to memory (after the first one) will collide with the previous access and will have to wait for the 6-clock-cycle bank busy time.
  - The load will take 391 (=12+1+6*63) cycles, 6.1 clocks per element

Memory bank conflicts will not occur within a single vector memory instruction if
- The stride and number of banks are relatively prime with respect to each other, and
- There are enough banks to avoid conflicts in the unit stride case

With 64 banks, a stride of 32 will stall on every other access, rather than every access.
With 256 banks, a stride of 32 will stall on every 8 access.

Increasing the number of memory banks to a number greater than the minimum to prevent stalls with a stride of length 1 will decrease the stall frequency for some other strides.

And...

Programming for avoiding conflicts whenever possible
Chaining
- The concept of forwarding extended to vector registers
- The results from the first functional unit in the chain are forwarded to the second functional unit.

Example:

MULV.D V1,V2,V3
ADDV.D V4,V1,V5

A sustained rate (ignoring start-up) of two floating-point operations per cycle. Total running time: (Vector length) * (Start-up Time_{ADD}) + (Start-up Time_{MULV})
Two reasons why higher levels of vectorization are not achieved

- Presence of conditionals (if statements) inside loops
  - Programs that contain if statements in loops cannot be run in vector mode using the techniques we have discussed so far.

- Use of sparse matrices
  - Sparse matrices cannot be efficiently implemented using any of the capabilities we have seen so far.

How can we handle conditional execution?

Example

```
    do 100 i = 1, 64
   100 if (A(i) .ne. 0) then
          A(i) = A(i) - B(i)
       endif
    continue
```

But, if the inner loop could be run for the iteration for which A(i) .ne. 0, then the subtraction could be vectorized.

Vector-mask control

Use a Boolean vector of length MVL to control the execution of a vector instruction just as conditionally executed instructions use a Boolean condition to determine whether an instruction is executed.

When the vector-mask register is enabled, any vector instructions executed operate only on the vector elements whose corresponding entries in the vector-mask register are 1.

Example

```
    LV Y1, Ra
    LV Y2, Rb
    LD F0, 0
    SNEYS, D Y1, F0
    SUBY, D Y1, V1, V2
    CVN
    SV Ra, V1
```

;load vector A into Y1
;load vector B
;load FF zero into F0
;sets VM(i) to 1 if V1(i) .ne. F0
;subtract under vector mask
;set the vector mask to all 1s
;store the result in A
Sparse Matrices

The elements of a vector are usually stored in some compacted form and then accessed indirectly.

Example: A sparse vector sum on the arrays A and C, using index vector K and M to designate the nonzero elements of A and C.

\[
\text{do } 100 \text{ i=1,n} \\
100 \quad A(K(i)) = A(K(i)) + C(M(i))
\]

Scatter-Gather Operations using index vectors

- Support moving between a dense representation (i.e., zeros are not included) and normal representation (i.e., the zeros are included) of a sparse matrix

A gather operation takes an index vector and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector.

- The result is a non-sparse vector in a vector register.

After these elements are operated on in dense form, the sparse vector can be stored in expanded form by a scatter store, using the same index vector.

Example

\[
\text{do } 100 \text{ i=1,n} \\
100 \quad A(K(i)) = A(K(i)) + C(M(i))
\]

Scatter-gather

- \[L_\text{Y} \quad \text{Vr, Rk} \quad \text{load K}\]
- \[L_\text{YI} \quad \text{Vr, (Rr+Vr)} \quad \text{load A(K(i))}\]
- \[L_\text{Y} \quad \text{Vn, Rm} \quad \text{load M}\]
- \[L_\text{YI} \quad \text{Vc, (Rc+Vc)} \quad \text{load C(M(i))}\]
- \[A_\text{D} \quad \text{Vr, Vr, Vc} \quad \text{add them}\]
- \[S_\text{VI} \quad \text{(Rr+Vr), Vr} \quad \text{store A(K(i))}\]

\(R_r, R_c, R_k,\) and \(R_m\) contain the starting addresses of the vectors

\(L_\text{Y}:\) load vector indexed

\(S_\text{VI}:\) store vector indexed
- Combination of parallel and pipelined functional units
- Each lane contains one portion of the vector-register file and one execution pipeline from each vector functional unit
- Each vector functional unit executes vector instructions at the rate of one element group per cycle using multiple pipeline, one per lane.
- Inter-lane wiring is only required to access main memory
<table>
<thead>
<tr>
<th>Bank 0</th>
<th>Stride-32 access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 8 16 24 32 40 48 56 64 ⋯</td>
</tr>
<tr>
<td>Bank 1</td>
<td>Stride-8 access</td>
</tr>
<tr>
<td>1</td>
<td>1 9 17 25 33 41 49 57 65 ⋯</td>
</tr>
<tr>
<td>Bank 2</td>
<td>Stride-1 access</td>
</tr>
<tr>
<td>2</td>
<td>2 10 18 26 34 42 50 58 66 ⋯</td>
</tr>
<tr>
<td>Bank 3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3 11 19 27 35 43 51 59 67 ⋯</td>
</tr>
<tr>
<td>Bank 4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4 12 20 28 36 44 52 60 68 ⋯</td>
</tr>
<tr>
<td>Bank 5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5 13 21 29 37 45 53 61 69 ⋯</td>
</tr>
<tr>
<td>Bank 6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6 14 22 30 38 46 54 62 70 ⋯</td>
</tr>
<tr>
<td>Bank 7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7 15 23 31 39 47 55 63 71 ⋯</td>
</tr>
</tbody>
</table>

A stride of a prime number to # of banks does not cause bank conflict!