Chapter 3 Pipelined Vector Processors

In the early 1970s, computer designers turned to pipelined vector processors to achieve higher performance. The pipelined vector processors were a natural evolution from the high performance serial machines of the day. To increase performance, the designers added special machine instructions for operating on vectors of numerical values which exploited the available parallelism in pipelined functional units.

In 1973, Control Data Corporation shipped the first vector processor called the STAR 100. Though historically first, the STAR 100 was not completely successful due partly to the use of magnetic-core memory, an old technology which had been surpassed by semiconductor memory. Furthermore, the best serial computers, i.e., the CDC 7600 and the IBM 360/195, had much faster arithmetic units for scalar operations and, therefore, the STAR 100 was slower except on very long vectors.

In 1976, the highly successful Cray-1 was introduced. Unlike the STAR 100, the Cray-1 could perform fast scalar arithmetic and even faster arithmetic on vectors. Since the Cray-1 is a classic example of a pipelined vector processor, this chapter will use the Cray-1 as its primary example. We will contrast the register-to-register style of the Cray-1’s vector operations with the memory-to-memory style of the CDC 205. The CDC 205 is a re-engineered version of Control Data’s STAR 100. We will discuss both the need of vectorizing compilers in order to exploit the parallelism introduced by the pipelined vector processors, and the changes in programming style introduced by these vector processors.

3.1 The Cray-1 Supercomputer

In 1972, Seymour Cray left Control Data Corporation to start his own company, Cray Research, Inc., with the aim of producing the fastest computer in the world. In the extraordinarily short time of four years, the Cray-1 computer was designed and built (1976). The Cray-1 follows the evolutionary trend of Seymour Cray’s CDC 6600 and CDC 7600. Where the ten serially organized functional units of the CDC 6600 had been replaced by eight pipelined functional units in the CDC 7600, Seymour Cray increased the number of pipelined functional units in the Cray-1 to twelve. We will first discuss the physical characteristics of the machine and then its architecture.
3.1.1 Physical Characteristics of the Cray-1

The Cray-1 has been called “the world’s most expensive love-seat.” The machine comprises a central 4.5 foot diameter cylindrical column 6.5 feet high, surrounded by the circular “love-seat” bringing the diameter at floor level to about 9 feet. The most striking feature of the Cray-1 is its small size. This is well illustrated in Figure 3.1. In computer design, smaller means faster. The greater the separation of components, the longer the time taken for a signal to pass between them. A cylindrical shape was chosen for the Cray-1 in order to keep wiring distances short. The maximum signal path is 3.5 feet. The cabinet for the CPU and memory is composed of 12 wedge-like columns in a 270 degree arc. "This leaves room for a reasonably trim individual to gain access to the interior of the machine."24

The love-seat around the base of the Cray-1 encloses the power supplies and some plumbing associated with the Freon cooling. Two 25-ton compressors comprise the Freon cooling system and are located externally to the computer room. The total power consumption of the machine is 128 kW, most of which generates heat that must be removed by the Freon cooling system. The Cray-1 cost about five million dollars and was affordable only by large national scientific laboratories where absolute performance at any cost was important.

3.1.2 Architecture of the Cray-1

An architectural diagram of the Cray-1 is given in Figure 3.2.

---

3.1 The Cray-1 Supercomputer

3.2 Architecture of the Cray-1

Functional Units

The Cray-1’s twelve independent functional units -- organized in four groups: Vector, Floating-point, Scalar and Address -- are shown on the right side of Figure 3.2. Each functional unit is pipelined and can deliver a result every clock period. Note that all the functional units use parallelism in the form of pipelining and all can operate at the same time (replication parallelism). Note further the absence of a divide unit in the Cray-1. Since floating point division cannot be pipelined, the Cray-1 performs a floating-point division by performing a reciprocal approximation followed by a multiply as illustrated by the following mathematical identity:

\[
\frac{x}{y} = \frac{1}{y} \ast x
\]

This approach has the advantage that reciprocal approximation can be pipelined.
The Pop./LZ (population and leading zero count) functional unit counts the number of ones or the number of leading zeros in an argument.

Registers

The Cray-1 has over 4 Kbytes of high speed (6 nanoseconds) register storage organized as shown in the middle vertical section of Figure 3.2. Since the Cray-1 is a register-to-register architecture, the twelve functional units use operands from and store results to the A, S and V registers. The B and T registers are used as auxiliary storage for the A and S registers, respectively.

The eight 24-bit A registers are primarily used as address registers for memory references and as index registers, but also are used to provide values for shift counts, loop control, and channel I/O operations. The Address functional units use the A registers to operate on 24-bit integers to modify addresses by adding, subtracting and multiplying A register quantities. The results are returned to the A registers to be used by the Instructional Unit in the instruction and data fetch stages of instructions.

The eight 64-bit S registers are used for scalar operations. Logical, shift, fixed-point (i.e., 64-bit integer) and floating-point operations may be performed on the values in the S registers. Also, scalar quantities involved in vector operations are held in the S registers.

The eight 64-element V registers provide vector operands to and receive vector results from the Vector and Floating-point functional units at the rate of one result per clock period. Each element of a V register holds a 64-bit quantity. A vector of up to 64 values may be held in a V register. The operation of the V registers will be discussed in Section 3.1.3.

The sixty-four 24-bit B registers are used as auxiliary storage for the A registers. The transfer of a 24-bit operand between A and B registers requires only one clock period. A block of data in the B register may be transferred to or from memory at the rate of one word (two 24-bit items) per clock period. Thus, it is possible to prefetch data from memory into the B registers before the A registers need it, e.g., just prior to a subroutine call.

The sixty-four 64-bit T registers are used as auxiliary storage for the S registers. The transfer of an operand between the S and T registers requires one clock period. A block of data in the T registers may be transferred to or from memory at a rate of one word per clock period.

Memory

The Cray-1’s memory of 1 Megaword is organized into 16 banks which allow 16-way interleaving of memory accesses. Each word in memory is 64-bits plus 8 additional bits for error detection and correction. The additional 8 bits allow for error correction if one bit of the 72 is bad and for error detection if two bits of the 72 are bad. A word may be fetched from or written to a bank in four clock periods (50 nanoseconds).

Fetching Instructions

Instructions are expressed in either one or two 16-bit parcels. The four banks of Instruction Buffers (IB) registers are used to buffer parcels from memory to the instructional unit. Each bank may hold 64 16-bit parcels. The four banks may each read a 64-word from a memory bank in four clock periods for a maximum transfer rate of 80 Megawords per second, assuming the Instruction Buffers are retrieving instructions from different memory banks. This equals four parcels a clock period. Since the computational section processes instructions at a maximum rate of one parcel per
Moving Data

In a high performance computer, overall system performance depends critically on how fast the computer moves data around. This is especially true of a computer with an expensive high speed computation unit as in the Cray-1. The Cray-1 may access memory in 50 nanoseconds or four clock periods. To speed up the effective access rate, a memory access can be made in parallel from each of sixteen memory banks for a potential of 320 Megawords per second. However, the Cray-1 does not have the data paths to fully utilize this memory bandwidth for operands. The data paths between memory and the A, B, S, T and V registers allow a maximum of only 80 Megawords per second and, furthermore, the flow of data must be in the same direction. For example, the data path cannot load information from memory to a V register and store another V register to memory at the same time.

Let us explore whether the memory bandwidth (transfer rate) is fast enough for the computation section. A typical arithmetic operation requires three memory accesses for its operands. For example, to add two arguments requires two loads and one store. The maximum computing rate of the Cray-1 is 160 MFLOPS (80 million floating-point adds and 80 million floating-point multiplies). Therefore, we need a memory bandwidth of 3 words/operation x 160 million operations = 480 Megawords per second.

As we have seen, the Cray-1 has a maximum memory bandwidth of only 80 Megawords per second. This low register-to-memory bandwidth has been a major criticism of the Cray-1. However, in a register-to-register architecture, many of the arguments will reside in a register and will help alleviate the problems of this register-to-memory bottleneck.

3.1.3 Vector Operations on the Cray-1

To illustrate a typical vector instruction on a register-to-register style vector computer, we will discuss the floating-point adder on the Cray-1. Recall the four stage pipelined floating-point adder of section 1.10.1.

![Four-Stage Pipeline of Section 1.10.1](image)

Fig. 3.3 The Four-Stage Pipeline of Section 1.10.1

In that section, we demonstrated that the asymptotic speedup was four or, in general, the number of stages in the pipeline. However, in order to approach a speedup of four, we had to keep the pipeline busy. In the Cray-1, we keep the functional unit pipelines busy by doing operations on vectors of numerical values. The Cray-1 has eight vector registers (V₀ - V₇), each of which may hold sixty-four 64-bit floating-point numbers. One machine instruction can add a vector register of 64 values to another vector register of 64 and place the result in a third vector register. The Cray-1 uses a six-stage pipeline for its floating-point adder.
For example, assume we are performing a vector add of 64 values in $V_0$ and $V_1$ and storing the results in $V_2$. In the first clock period, the operands at location 0 in the two vector registers $V_0$ and $V_1$ are fed to the first stage of the pipeline. After six clock periods, the first result is placed in location 0 of $V_2$. Every clock period after that one result is produced and placed in $V_2$. How much faster is the parallel (pipelined) version over the serial one?

It takes $6t_c$ to perform a floating-point add where $t_c$ is the time for a machine cycle or clock period (12.5 nanoseconds on the Cray-1).

$$t_{\text{serial}} = \text{number of pairs} \times \text{time for one add} = 64 \times 6t_c = 384t_c$$

For the pipeline, it takes $6t_c$ for the first pair and one $t_c$ for each of the rest. Let $n$ be the number of pairs of values to be added and $k$ be the number of stages in the pipeline.

$$t_{\text{pipeline}} = kt_c + (n - 1)t_c = (k + n - 1)t_c = (6 + 64 - 1)t_c = 69t_c$$

speedup $= \frac{384t_c}{69t_c} = 5.6$

Therefore, for a vector length of 64, the pipelined adder hardware can increase the performance by a factor of 5.6 over scalar. However, this is not the complete story. Consider the following FORTRAN code where $N$ is 64.

```
DO 10 I = 1, N
    C(I) = A(I) + B(I)
10 CONTINUE
```

In a traditional scalar processor, a FORTRAN compiler would generate something close to the following machine language equivalent:

```
STORE 1 into I
LABEL1: BRANCH to LABEL2 if I > N
LOAD A[I]
ADD B[I]
```
3.1 The Cray-1 Supercomputer

On the Cray-1, the FORTRAN compiler generates the equivalent code with four vector instructions.

```
VECTOR-LOAD V0 with A, 64
VECTOR-LOAD V1 with B, 64
VECTOR-ADD V2 ← V0 + V1, 64
VECTOR-STORE V2 into C, 64
```

The first VECTOR-LOAD instruction loads 64 values into the vector register V0 starting at address A in memory. The VECTOR-ADD instruction is the one pictured in Figure 3.4. In the serial case, 385 (i.e., 1 + 64 * 6) instructions must be fetched and decoded compared to 4 in the vectorized case. This overhead in fetching and decoding instructions as well as the overhead in software loop control significantly slows down the serial case. However, one pays for the fast execution of the vector instructions on a Cray-1 style machine by needing extra, expensive hardware.

### 3.1.4 Chaining on the Cray-1

To increase performance of certain combinations of vector operations, the Cray-1 incorporates the technique called chaining. In chaining, the resultant stream of one functional unit is fed directly to another functional unit rather than stored in a V register. For example, consider the following FORTRAN code with a vector add followed by a vector multiply.

```
DO 10 I = 1, 64
   A(I) = (B(I) + C(I)) * D(I)
10 CONTINUE
```

Without chaining, the vector add would be required to completely add all 64 elements and store them in a V register. With chaining, the result of the floating-point add functional unit is fed directly to the multiply functional unit as shown in Figure 3.5.

![Fig. 3.5 Chaining of Vector Add to Vector Multiply](image)

After six clock periods, the first result of the add functional unit is fed to the multiply unit. After another seven clock periods (the multiplier has a seven stage pipeline), the first result is stored in the V3 register. Chaining is almost twice the speed compared to without chaining as shown in Figure 3.6.
Notice that by chaining together a six-stage pipeline with a seven-stage one, we have effectively a thirteen-stage pipeline with an asymptotic speedup of thirteen. Therefore, for increased performance, computer designers strive to increase the length of computer pipelines by chaining several together. For example, to increase the performance of the Cray-1’s memory-to-register transfers, portions of the transfer are chained together into an eleven-stage pipeline. This facilitates a fast move of a block of data from memory to, for example, the V registers. Assuming the pipeline is kept full, the asymptotic speedup is eleven for moving data. However, a designer must weigh the benefits of such a pipeline with the time it takes one value to flow through the pipeline (pipeline latency). For single values, we want the pipeline latency to be as short as possible. For example, if the pipeline is empty, it takes eleven clock periods to obtain a value by way of the pipeline.

Observe in Figure 3.6 that after 13 clock periods, both the add and multiply functional units produce a result every clock period for fifty-six clock periods. Notice that only through chaining of a vector add and vector multiply can the Cray-1 achieve the stated peak performance of 160 MFLOPS (80 million floating-point adds and 80 million floating-point multiplies per second), and even then only in bursts. Now it should be clear why the peak performance of the Cray-1 is rarely achieved by typical application programs. If we consider typical FORTRAN programs, the chaining of a vector add and a vector multiply is a relatively rare event.

### 3.1.5 Reasons for the High Performance of the Cray-1

Remember, we are using the Cray-1 as a representative example of a pipelined vector processor. Since later vector processors borrow many of the same architectural ideas, it is worthwhile to explore several general principles used in the Cray-1 to achieve high performance.

First, for a high performance machine, the computer designers must devise a carefully balanced design of all the major parts including fast memory, high bandwidth from memory to CPU and fast I/O, or else a major component becomes the bottleneck. This balance is achieved to a high degree in the Cray-1; however, we have seen that the memory-to-register transfer can be a bottleneck.

A second principle used in the Cray-1 was the incorporation of parallelism in many aspects of the machine. Parallelism by replication is seen in the sixteen independent banks of memory, the twelve independent functional units and the multitudes of registers. Parallelism by pipelining is evident in the functional units, the instruction stream and the memory-to-register transfer.
A third principle is the use of buffering between agents to allow both to work as fast as possible. In a consumer-producer situation, the producer places data in the buffer then continues doing work. When the consumer is ready, it uses the data from the buffer. Buffering minimizes wasteful waiting by either the consumer or producer. In the Cray-1, the T registers are buffers between memory and the S registers. The B registers are buffers between memory and the A registers. By using the memory-to-register pipeline to move blocks of data from memory to the V, T and B registers, data can be available when needed by the functional units. Therefore, the functional units can be kept busy and rarely need to wait for memory accesses. Also, instructions are buffered by the four banks of instruction buffers.

The fourth principle of high performance is to use a fast clock. This implies short signal path and expensive technology.

### 3.2 More Recent Cray Supercomputers

This section will discuss the Cray supercomputers manufactured since the Cray-1. The purpose is not to provide a detailed description of each machine but to introduce innovative ideas of parallel processing used in the machines. The main characteristics of the Crays are summarized in Figure 3.7.

**Cray-1S**

The Cray-1S, first shipped in 1979, was Cray Research’s reaction to several shortcomings of the Cray-1. First, though 1 Megaword (8 Megabytes) was a lot of primary memory in 1976, the scientific computing community wanted more. In response, the Cray-1S allowed four times the capacity with 4 Megawords. Much like junk in an attic, scientific programs tend to grow to fill all available space. As the chart in Figure 3.7 shows, this trend of larger and larger memories has continued for the last decade and a half. Second, the I/O facilities of the Cray-1S were greatly enhanced over the Cray-1.
### Table: Characteristics of Cray Supercomputers Over Time

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Cray 1</th>
<th>Cray 1S</th>
<th>Cray X-MP/2</th>
<th>Cray X-MP/48</th>
<th>Cray 2</th>
<th>Cray Y-MP</th>
<th>Cray C90</th>
<th>Cray 3&lt;sup&gt;25&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor clock</td>
<td>12.5 ns</td>
<td>12.5 ns</td>
<td>9.5 ns</td>
<td>9.5 ns&lt;sup&gt;26&lt;/sup&gt;</td>
<td>4.1 ns</td>
<td>6 ns</td>
<td>4 ns</td>
<td>2 ns</td>
</tr>
<tr>
<td>Number of processors</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>16&lt;sup&gt;27&lt;/sup&gt;</td>
<td>16</td>
</tr>
<tr>
<td>Main memory</td>
<td>1 Mw</td>
<td>4 Mw</td>
<td>4 Mw</td>
<td>16 Mw</td>
<td>256 Mw</td>
<td>32 Mw</td>
<td>256 Mw</td>
<td>512 Mw</td>
</tr>
<tr>
<td>Memory cycle time</td>
<td>50 ns</td>
<td>50 ns</td>
<td>38 ns</td>
<td>38 ns</td>
<td>250 ns&lt;sup&gt;28&lt;/sup&gt;</td>
<td>15 ns</td>
<td>15 ns</td>
<td>25 ns</td>
</tr>
<tr>
<td>Memory bandwidth (data)</td>
<td>80 Mw/s 80 Mw/s</td>
<td>630 Mw/s</td>
<td>1.2 Gw/s</td>
<td>1 Gw/s</td>
<td>32 Gw/s</td>
<td>1 Gw/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory interleaved</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td>I/O system</td>
<td>enhanced over Cray 1</td>
<td>SSD 1024 Mb 50 microsec</td>
<td>20 times over Cray 1</td>
<td>13.6 GB/s</td>
<td>1 GB/s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peak performance (MFLOPS)</td>
<td>160</td>
<td>160</td>
<td>470</td>
<td>940&lt;sup&gt;29&lt;/sup&gt;</td>
<td>1952</td>
<td>2667</td>
<td>16000</td>
<td>8000</td>
</tr>
<tr>
<td>Sustained rate LINPACK Best&lt;sup&gt;30&lt;/sup&gt; (MFLOPS)</td>
<td>110</td>
<td>110</td>
<td>426</td>
<td>822</td>
<td>1406</td>
<td>2144</td>
<td>9715</td>
<td></td>
</tr>
<tr>
<td>Typical rate LINPACK&lt;sup&gt;31&lt;/sup&gt; (MFLOPS)</td>
<td>27</td>
<td>27</td>
<td>143</td>
<td>178</td>
<td>62</td>
<td>275</td>
<td>479</td>
<td></td>
</tr>
<tr>
<td>Cost&lt;sup&gt;32&lt;/sup&gt;</td>
<td>$4.5M</td>
<td>$10M</td>
<td>$25M</td>
<td>$20M</td>
<td>$25M</td>
<td>$30M</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 3.7 Characteristics of Cray Supercomputers Over Time**

<sup>25</sup> All but the Cray 3 were manufactured by Cray Research, Inc. The Cray 3 is manufactured by Cray Computer Co.

<sup>26</sup> In August 1986, the clock period was reduced from 9.5 nanoseconds to 8.5 nanoseconds.

<sup>27</sup> The Cray Y-MP C90 has two vector pipelines per processor.

<sup>28</sup> In order for the Cray 2 to have a large main memory of 256 Megawords, Seymour Cray used a slower main memory and a high speed local memory of 16 Kwords with 16 nanosecond access time in each processor.

<sup>29</sup> Peak performance with 8.5 ns clock.

<sup>30</sup> “Best Effort” rating on LINPACK benchmark where changes to software are permitted and the order of the linear equations is 1000 [Dongarra, 1992].

<sup>31</sup> Rating on the LINPACK benchmark with no changes to the software and the order of the linear equations is 100 [Dongarra, 1992].

<sup>32</sup> Approximate cost for a fully configured machine.
Cray X-MP

A competitive group within Cray Research, Inc. headed by Steve Chen (not Seymour Cray) designed the Cray X-MP. Demonstrating the advances in packaging and in technology, the group managed in 1982 to fit a two-CPU version (X-MP/2) in the same cabinets as the Cray-1.

The Cray X-MP has a Cray-1-like architecture which fixes a major deficiency present in the Cray-1. In place of the Cray-1’s single data path, the Cray X-MP has three 64-bit data paths from memory to the registers. For example, two load operands from memory into V registers and one store operand from a V register to memory are allowed at the same time. While the Cray-1 transfers data in only one direction, the Cray X-MP can transfer data in both directions, e.g., perform a load of a vector at the same time as a store of a vector. Because the data paths are independent, two 64-bit load operands and one 64-bit store operand can be transferred per clock period, giving a data transfer rate of 315 Megawords per second per CPU compared with 80 Megawords per second on the Cray-1. Therefore, the Cray X-MP eliminated the memory-to-register bandwidth bottleneck of the Cray-1.

The three new data paths of the Cray X-MP allow what Cray Research calls “flexible chaining.” Consider the following FORTRAN code with a vector add.

```fortran
 DO 10 I = 1, 64
   C(I) = A(I) + B(I)
10 CONTINUE
```

The above vector add is performed by four vector instructions on a Cray.

- VECTOR-LOAD V0 with A, 64
- VECTOR-LOAD V1 with B, 64
- VECTOR-ADD V2 ← V0 + V1, 64
- VECTOR-STORE V2 into C, 64

Let us compare these four vector instructions executed on the Cray-1 with flexible chaining on the Cray X-MP. Because of the Cray X-MP’s three data paths, the two Loads and the Store can overlap each other and the computation. While in contrast, on the Cray-1 the second Load must wait for the first Load to finish and the Store must wait for the second Load to complete. Also, on the Cray-1, the computation of the pipelined functional units may overlap a load but not a store.

![Fig. 3.8 The Computation on Cray-1 (No Flexible chaining).](image-url)
Flexible chaining is another example of chaining pipelines together to significantly increase performance, in this case, 71 versus 198 clock periods.

In 1984, the use of high density integrated chips allowed Cray Research to build a four-CPU model (Cray X-MP/4) housed in the same cabinets as the Cray-1. This doubled the performance. In 1986, in order to squeeze out even more performance, Cray Research shortened the clock period of the Cray X-MP system clock from 9.5 nanoseconds to 8.5 nanoseconds.

Another major improvement of the Cray X-MP over the Cray-1 was the Input/Output (I/O) system. We will examine the Cray X-MP’s sophisticated I/O system as an example I/O system of today’s supercomputers. With high performance computing, a machine must be able to move data in and out of memory to I/O devices fast; otherwise, the machine becomes I/O bound and the expensive CPU is left idle.

The Cray X-MP/4 is connected by way of one or two 1024 Mbytes/second channels to a Cray SSD. The SSD (Solid State Device) is a solid state MOS storage device ranging from 64 to 1024 Mbytes, which can be used in place of disk storage for the data of very large user programs and by the operating system for temporary storage. The SSD acts like a disk with an access time of less than 50 microseconds.

Through an I/O Subsystem (IOS), the Cray X-MP is connected to front-end computers, magnetic tape drives, disk storage units and fast network links. The IOS consists of two to four 16-bit I/O processors and an 8 Mword buffer memory. The first I/O processor handles communication with up to three front-end processors, e.g., an IBM mainframe or a DEC VAX. The second I/O processor handles communication between 16 high speed disk units and the Cray X-MP’s common memory via the buffer memory. The third and fourth I/O processors are optional and may be used to attach further disk units. The DD-49 disk units introduced in 1984 have a capacity of 1200 Mbytes and a 10 Mbytes/second transfer rate. All the CPUs of a Cray X-MP share the one or two 1250 Mbyte/second channels to the SSD and two 100 Mbytes/second channels to the I/O processors. Note the use of parallelism in off loading the I/O burden from the CPUs to the I/O processors.

Cray Research has an ongoing commitment to high-speed peripherals and fast network links. See the section on the Cray Y-MP C90 supercomputer for the latest characteristics in I/O systems.

A new development of the Cray X-MP over the Cray-1 was the use of multiple CPUs (an example of replication parallelism). The CPUs share a common memory. The two or four CPUs are coordinated by an intercommunication section which contains three clusters of common registers (five clusters on an X-MP/48) that may be accessed by all CPUs for interprocessor communication and synchronization. There is also a common real time clock that allows a programmer to time program segments to the nearest clock period. Programming the multiple CPUs will be discussed in section 3.4.
Another consequence of multiple CPUs is a more complex memory system. Each CPU has four independent ports to memory which allows two loads, a store and an I/O operation to take place at the same time. The I/O port is reserved for the IOS and the SSD. The A, B and C ports are available for the data paths to the registers. The instruction registers must also share the four ports for fetching machine instructions. The four ports of a CPU are connected by a crossbar switch to a section of memory containing eight interleaving memory banks (sixteen interleaved memory banks in a four-CPU Cray X-MP/4). The crossbar switch allows any port to access any section of memory. Figure 3.10 shows the memory system for a two-CPU Cray X-MP/2. A four-CPU Cray X-MP/4 is similar with a crossbar switch for each CPU and sixteen interleaved memory banks per section. Contrast this diagram with the diagram in Figure 1.10. Extra hardware is needed to resolve conflicts at the port, memory section and memory bank levels. This sophisticated memory system achieves a high memory bandwidth of three words per clock period per processor or 1200 Megawords/second (for a 9.5 nanosecond clock).

In scientific computing, the scatter and gather operations occur often. The scatter operation can be defined by the following pseudo-code:

```pseudocode
for I := 1 to N
    X[Index[I]] := Y[I]
endfor
```

which scatters values of $Y$ arbitrarily throughout main memory according to the values in the Index array. Conversely, the gather operation collects the scattered elements of $X$ and places them in the ordered array as defined by the following:

```pseudocode
for I := 1 to N
    Y[I] := X[Index[I]]
endfor
```

Prior to 1984, the early Crays had no special hardware for scatter and gather operations. Such operations had to be done as scalar loops, with disappointing performance. In 1984, the newly introduced four-CPU Cray X-MP had special hardware and instructions for the implementation of scatter and gather operations with an associated increase in speed. Cray Research introduced hardware scatter and gather partly due to market pressure from its chief competitor, the CDC 205, which had hardware scatter and gather since 1981.
Cray-2

While Steve Chen and his group at Cray Research were designing the X-MP/48, Seymour Cray, still at Cray Research, was designing the Cray-2. Shipped in 1985, the Cray-2 is a new architecture with a new instruction set and operating system (Unix based). The Cray-2 is a four-CPU pipelined vector machine all contained within a cylindrical cabinet 4 feet high and 4.5 feet in diameter. The compression in size is remarkable -- in effect, a four-CPU Cray X-MP plus its I/O system and SSD have been shrunk to a third or a quarter of their present size and placed in a single container. The small size allows a fast clock (4.1 nanoseconds) and short signal paths (longest wire is 16 inches).

The Cray-2 uses a brand new cooling technology called liquid immersion cooling. Whereas the older Crays used metal heat sinks and Freon piped through them to draw off the heat from the circuit boards, all the circuit boards in the Cray-2 are totally immersed in a bath of clear, electrically inert fluorocarbon liquid. The fluorocarbon liquid is circulated by a pump and passed through a chilled water heat exchanger to extract the heat.

In the mid-1980s, computational scientists were running many large programs which exceeded the capacities of supercomputers’ main memory. They had to resort to manually transferring very large datasets in and out of memory to and from a fast disk. Therefore, to relieve this burden, Seymour Cray wanted the Cray-2 to have a very large common memory of 256 Megawords. However, in order to achieve such a large memory, he had to compromise the memory speed and use, as he calls it, “PC memory” with an access time of 256 nanoseconds. In an attempt to regain the loss in main memory speed, Seymour Cray replaced the B and T intermediate registers of the Cray-1 with 16 Kwords of local memory in each processor. This local memory can be accessed in four machine cycles (16 nanoseconds). The mismatch in this two-level memory hierarchy of 256-nanosecond main memory and 16-nanosecond local memory causes performance problems. For example, on many programs, the Cray-2 is memory bound, i. e., the performance is limited by memory and not by the functional units. This memory-to-CPU mismatch is the primary reason for the Cray-2’s poor performance on the LINPACK benchmark (62 MFLOPS) shown in Figure 3.7.

The architecture of the Cray-2 is similar to the Cray-1; however, the ability to chain together a succession of vector instructions, which is an important feature of the Cray-1 and Cray X-MP, is not available on the Cray-2.

Cray Y-MP

Shipped in 1987, the Cray Y-MP has a similar architecture to the Cray X-MP. A major difference is the availability of 32-bit as well as 24-bit addressing which allows main memory to be larger than 16 Megawords. Compared to the Cray X-MP, the Cray Y-MP has a faster machine cycle time (6 nanoseconds) and more processors (eight).

Cray Y-MP C90

Introduced in November, 1991, the Cray Y-MP C90 is Cray Research’s latest model high end supercomputer. The C90 enhances the Y-MP line with twice the number of processors (16), much larger main memory (256 Megawords) and a faster clock (4 nanoseconds).

A new development is the introduction of two vector pipelines for each processor. This allows four floating-point operations per clock period for each processor. Notice that a fully configured CRAY Y-MP C90 features 64-way replication parallelism with sixteen processors each with two vector pipes and two functional units (add and multiply) per pipe.
3.2 More Recent Cray Supercomputers

Each CPU has four ports to memory, each of which may read a double word (128 bits) in one clock cycle. That is, each CPU can read eight words in one clock cycle for a total memory bandwidth of 32 Gigawords/second.

The Cray Y-MP C90 features up to 13.6 Gigabytes/second of aggregate I/O bandwidth. The I/O subsystem supports up to 16 I/O clusters (processors) for connection to disk storage units, tape units and networks. Each I/O cluster can be configured with 16 DD-60 disk drives to deliver up to 320 Mbytes/second performance. The DD-60 disk drives each hold 2 Gigabytes and have a transfer rate of 24 Mbytes/second. The system supports up to four 1800 Mbytes/second channels to the SSD which may hold up to 2048 Mwords. Cray Research provides support for the very high-speed networks such as High-Performance Parallel Interface (HIPPI) (100 Mbytes/second) and Fiber Distributive Data Interface (FDDI) for connection to other computers.

Cray 3

In 1989, Seymour Cray left Cray Research to found another start-up company, Cray Computer Co., where he pursues the gallium arsenide (GaAs) technology integral to the Cray-3. Cray has demonstrated that gallium arsenide integrated circuits can be fabricated to execute three times faster than silicon. The Cray-3 is to be delivered in 1992.

The Cray-3 is essentially a GaAs version of Seymour Cray’s Cray-2. However, the Cray-2 memory mismatch has been solved by using much faster main memory (512 Mwords of 25 nanosecond memory). Also, the computing imbalance between scalar and vector computing in the Cray-2 has been removed. Scalar speed on one processor is four times that of a Cray-2 processor.

3.3 The CDC 205 Supercomputer

In 1979, Control Data Corporation reacted to Cray Research’s newly introduced Cray-1 by completely re-engineering the CDC STAR 100 and renaming the machine the CDC 203. In 1981, after further enhancements, the machine was named the CDC 205. A pipelined vector computer, the CDC 205, the main competitor to the Cray-1, had several interesting features worth studying.

In contrast to the register-to-register style architecture of the Cray-1, the CDC 205 is a memory-to-memory vector processor. That is, rather than move a vector from memory to registers where it is operated on by functional units, the CDC 205 moves the vector directly from memory to the functional unit and the resultant vector is stored back in memory. This has the major advantage that vectors may be very long (up to 65,535 64-bit values). On the Cray-1, to operate on two long vectors, the FORTRAN compiler would generate code to compute strips of 64 elements because of the 64-value size limitation of the V vector registers. For example, to add two 1000 element vectors, the Cray-1 would loop fifteen times around a body of code which includes two vector loads, a vector add and a vector store of 64-value strips (the process is called “strip mining”). Since 1000 is not evenly divisible by 64, the compiler would generate extra code to handle the last 40 elements. In contrast, the CDC 205 could perform the same vector add of two 1000 vector elements in one instruction.

The memory-to-memory style of architecture has several consequences. First, the memory system becomes a very critical part of the machine. Therefore, the CDC 205 has an impressive memory system. Its 4 Mwords of memory (64-bit words) are divided into four sections each with a 512-bit wide data path. A memory access is 512-bits or a superword – “sword” – (eight 64-bit words) and takes 80 nanoseconds (4 clock periods). Each of the four data paths into memory can either read or write at the maximum rate of one superword per clock period (20 nanoseconds). The high rate is obtained by interleaving 8 memory banks in each of the 1 Mword sections. The
memory bandwidth is therefore 400 Mwords/second per million words of memory. We note that
this is five times the data memory bandwidth of the Cray-1 (80 Mwords/second).

A second consequence of the memory-to-memory architecture is that only a unit vector
increment (or stride) is allowed, i.e., the values in the vector must be stored contiguously in
memory. This creates a major problem, because many scientific programs do not have a stride of
one. Consider the following FORTRAN code:

```fortran
REAL A(3, 4), B(3, 4), C(3, 4)
DO 10 I = 1, 3
   DO 20 J = 1, 4
      A(I, J) = B(I, J) + C(I, J)
   20 CONTINUE
10 CONTINUE
```

In FORTRAN, arrays are stored by columns. That is, the matrix A
\[
\begin{array}{cccc}
a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\
a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\
a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\
\end{array}
\]
will be stored as the first column \((a_{1,1}, a_{2,1}, a_{3,1})\), followed by the second column \((a_{1,2}, a_{2,2}, a_{3,2})\) and so forth. Matrix A is stored in memory as follows:

\[
\begin{array}{cccc}
a_{1,1} & a_{3,1} & a_{2,1} & a_{1,1} \\
a_{2,1} & a_{3,1} & a_{2,1} & a_{1,1} \\
a_{3,1} & a_{3,1} & a_{2,1} & a_{1,1} \\
a_{1,2} & a_{3,2} & a_{2,2} & a_{1,2} \\
a_{2,2} & a_{3,2} & a_{2,2} & a_{1,2} \\
a_{3,2} & a_{3,2} & a_{2,2} & a_{1,2} \\
a_{1,3} & a_{3,3} & a_{2,3} & a_{1,3} \\
a_{2,3} & a_{3,3} & a_{2,3} & a_{1,3} \\
a_{3,3} & a_{3,3} & a_{2,3} & a_{1,3} \\
a_{1,4} & a_{3,4} & a_{2,4} & a_{1,4} \\
a_{2,4} & a_{3,4} & a_{2,4} & a_{1,4} \\
a_{3,4} & a_{3,4} & a_{2,4} & a_{1,4} \\
\end{array}
\]

Now consider the sequence of indices of the above nested DO loops. The nested loops operate on
the \((1, 1)\) elements; then the \((1, 2)\) elements, the \((1, 3)\) elements, the \((1, 4)\) elements, the \((2, 1)\)
elements, etc., or every third address for a stride of three.

In this case, we can move the J DO loop inside the I DO loop (called a loop interchange) to
create a stride of one.

```fortran
REAL A(3, 4), B(3, 4), C(3, 4)
DO 20 J = 1, 4
   DO 10 I = 1, 3
      A(I, J) = B(I, J) + C(I, J)
   10 CONTINUE
20 CONTINUE
```
But in general, interchanging the DO loops won’t eliminate non-unit stride. To handle the non-unit stride situation, the CDC 205 provides several solutions. First, if the required data is not consecutively stored, the required elements can be selected by a vector mask, one bit for each word of the vector. The operation is then performed on elements for which the corresponding mask bit is a one. However, all elements of the consecutively stored vector must be read from memory even though only a small fraction may be operated upon. Second, if a non-unit stride vector is to be operated on many times, the selected elements of the vector can be “compressed” and stored as a new temporary vector. Subsequent operations may then be performed with better efficiency on the newly compressed vector. Third, the efficient gather and scatter operations (see page 71), which are implemented in the hardware as microcode, may be used to form new temporary contiguously stored vectors. Therefore, even though the CDC 205 operates effectively on very long vectors, a significant amount of time is spent manipulating vectors to overcome the unit-stride restriction.

In contrast to the unit-stride requirement of the CDC 205, the Cray-1 allows constant stride, i.e., a vector may have elements a constant distance apart in memory. One advantage of the register-to-register architecture of the Cray-1 is that memory can fetch elements in a pattern of constant stride and store them in a V vector register before they are operated on by the functional units. However, when Seymour Cray added the 16 Kwords of local memory to the Cray 2, the Cray 2 lost this important feature. On the Cray-2, vector references from local memory to the V registers must be with unit stride. This means vectors transferred from main memory to local memory must be massaged on the Cray 2 much as on the CDC 205, with a similar decrease in overall performance.

The CDC 205 has both a scalar unit and at least one vector unit which may operate in parallel. The machine may have one, two or four vector floating-point arithmetic units called pipes. If two successive vector instructions use different functional units within a pipe and contain one operand that is scalar, it is called a linked triadic (from Greek for three) operation. Several examples are:

\[
\text{vector} + \text{scalar} \times \text{vector} \\
(\text{vector} + \text{scalar}) \times \text{vector}
\]

For a linked triad, the output stream from the first functional unit is fed to the input of the second functional unit, much like chaining on the Cray-1 but more restrictive. While the CDC 205 requires at least one scalar argument, on the Cray-1 all three arguments may be vectors. With a linked triad, each pipe on the CDC 205 may perform a 64-bit floating-point addition and a multiply every clock period (20 nanoseconds) for a rate of 100 MFLOPS. The maximum performance on the CDC 205 is therefore 400 MFLOPS for linked triads in 64-bit arithmetic on a 4-pipe machine. It is 800 MFLOPS on 32-bit arithmetic.

Since the 4-pipe CDC 205’s peak performance of 400 MFLOPS was significantly higher than the Cray-1’s peak of 160 MFLOPS, it was a very attractive machine. However, the CDC 205 has a long vector startup time as indicated by Hockney’s \( n^{1/2} \) parameter being 100 (for a 2-pipe machine). That is, to overcome the large vector startup time, vectors had to be more than 100 elements in length to achieve half peak performance (100 MFLOPS for a 2-pipe machine). In contrast, the Cray-1 has an \( n^{1/2} \) of about 10. For the 2-pipe CDC 205 (peak 200 MFLOPS) to be faster than the Cray-1, the vector lengths have to be greater than 300 [Hockney, 1988].

On the CDC 205, performance can be severely degraded at short vector lengths (less than 100) and if vectors are not stored contiguously. For this reason, scientific programmers write software for the CDC 205 which employ long, contiguously stored vectors.

The successor to the CDC 205 is the ETA\(^{10}\) (the superscript 10 is part of the name) by ETA Systems, created as a subsidiary of Control Data Corporation in 1983. The ETA\(^{10}\) may be described as eight up-rated CDC 205 computers working from a large common memory with the
resultant peak performance of 10 GFLOPS. Although over 30 machines were sold, the company ran into financial problems and folded in 1989.

### 3.4 Influences of Pipelined Vector Processors on Programming

This section will explore the influences of the pipelined vector processors on compiler design as well as the activity of programming.

#### 3.4.1 Vectorization

While it is preferable to have vector constructs in high-level languages, most scientific programs written for pipelined vector computers have been written in FORTRAN 77, which lacks vector constructs. Here we will limit our discussion to programming in FORTRAN 77, later in the book, we will discuss the new FORTRAN 90 language which does have vector constructs.

Vectorization is the process of identifying blocks of code suitable for translation to vector instructions which use the high performance pipelined vector units of a vector processor, e. g., Cray-1 or CDC 205. Basically, there are two approaches to vectorization, allow the programmer to specify which blocks of code should be vectorized or allow the compiler to determine what to vectorize. Since it is difficult for the programmer to specify what should be vectorized, most manufacturers have designed vectorizing compilers to handle the task automatically.

An example of a vectorizing compiler is the Cray-1’s Cray FORTRAN Translator (CFT). The CFT FORTRAN compiler analyzes the innermost DO loops of the FORTRAN programs it compiles to determine whether vector processing methods can be applied to improve overall program efficiency. If such efficiency can be improved, the compiler generates a sequence of code containing vector instructions to drive the high speed vector functional units. To be vectorized, a DO loop must manipulate or perform calculations on the contents of one or more arrays and not have certain language features such as GOTO, I/O and subroutine CALL statements that inhibit vectorization. It is easy to vectorize the following block of code into a vector add.

```fortran
DO 10 I = 1, N
   X(I) = Y(I) + Z(I)
10 CONTINUE
```

Other times, however, the translation is much more difficult for the compiler. Consider the following code where the first seven elements of array A are 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 and 7.0, respectively.

```fortran
DO 10 I = 2, 7
   A(I) = A(I - 1)
10 CONTINUE
```

The results in array A differ depending on if one executes the code serially or in parallel as shown in Figure 3.11. The reader should convince himself or herself of this fact by doing the six assignment statements of the loop in parallel.

<table>
<thead>
<tr>
<th>Original Value</th>
<th>Performed Serially</th>
<th>Performed in Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1) 1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>A(2) 2.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>A(3) 3.0</td>
<td>1.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>
A(4) 4.0 1.0 3.0  
A(5) 5.0 1.0 4.0  
A(6) 6.0 1.0 5.0  
A(7) 7.0 1.0 6.0

Fig. 3.11 Table Showing Results Performed Serially or in Parallel

Clearly, the parallel version is wrong because after the first iteration, each iteration of the loop depends on the previous iteration, i.e., there exist important data dependencies. Therefore, a vectorizing compiler performs an extensive dependency analysis and if a dependency is found it will not vectorize the loop.

The following code does not have a dependency and can be vectorized.

```
DO 10 I = 1, 99
   A(I) = A(I + 1)
10 CONTINUE
```

Sometimes, a vectorizing compiler can not tell if the code can be vectorized or not. For example, if \( J \) is positive, the following code can be vectorized.

```
DO 10 I = 1, 99
   A(I) = A(I + J)
10 CONTINUE
```

In this case, most vectorizing compilers will print a dependency message to inform the programmer of a possible dependency, then the programmer may insert a compiler directive to vectorize the loop. Of course, if \( J \) is negative and the compiler is told to vectorize the loop, the program produces incorrect results.

On pipelined vector computers such as the Cray-1, it is highly desirable to vectorize as much of the program as possible. Therefore, programmers manipulate the FORTRAN code to increase the potential of vectorization (called tuning the code). The CFT FORTRAN manual gives the following general guidelines for vectorization:\(^{33}\)

1. Keep subscripts simple and explicit, do not use parentheses in subscripts.
2. Do not use GOTO or CALL statements inside DO loops.
3. Use the Cray FORTRAN intrinsic functions where appropriate as they have already been tuned for vectorization.
4. Rewrite large loops containing a few unvectorizable statements as two or more loops, one or more of which will vectorize.

For an example of the last guideline, consider the following non-vectorizable code (from [Quinn, 1987]):

```
DO 10 I = 1, N
   X(I) = Y(I - 1)
   Y(I) = 2 * Y(I)
10 CONTINUE
```

However, the following set of two DO loops is functionally equivalent and both are vectorizable.

```
DO 10 I = 1, N
   Y(I) = 2 * Y(I)
10  CONTINUE
DO 20 I = 1, N
   X(I) = Y(I - 1)
20  CONTINUE
```

Sometimes, there is more than one way to vectorize a block of code and the compiler must determine which way has the potential for the highest performance. For example, it is better for the length of the vector to be as long as possible. Many vectorizing compilers will perform a loop interchange source transformation on the following code:

```
DO 10 I = 1, 100
   DO 20 J = 1, 10
      A(I, J) = B(I, J) * C(I, J)
   20 CONTINUE
10 CONTINUE
```

to the following code with a longer vector in the inner loop.

```
DO 20 J = 1, 10
   DO 10 I = 1, 100
      A(I, J) = B(I, J) * C(I, J)
   10 CONTINUE
20 CONTINUE
```

Because of this potential of loop interchange by a vectorizing compiler, programmers should avoid placing GOTO and CALL statements in all DO loops not just the inner most ones.

Vectorizing compilers have become very sophisticated in their attempts to vectorize code. See Michael Wolfe’s book on *Optimizing Supercompilers for Supercomputers* [Wolfe, 1989] for more detail. The CFT compiler is enhanced all the time, particularly, the optimization phase of the compiler which vectorizes the code. For example, in 1978, CFT on the Cray-1 was rated at 4.7 MFLOPS on the Lawrence Livermore Loops benchmark. In 1984 on the same benchmark and machine, the compiler generated code which performed 11.1 MFLOPS.

### 3.4.2 Programming Multiple CPUs

As an example of programming on multiple CPUs, we will discuss the approach used on the Cray computers.

With the introduction of multiple CPUs on the Cray X-MP in 1984, Cray Research provided system routines to allow the CPUs to cooperate in the solution of single user programs. This facility is called *multi-tasking* by Cray. On the Cray X-MP and later model Crays, the unit of work, called a task, that is handed for execution on a CPU is a FORTRAN subroutine, preferably one that must compute for a while. A task is started by a call to the TSKSTART system routine.

```
CALL TSKSTART(taskid, name of subroutine, list of parameters)
```

Effectively a *fork* operation, the call to TSKSTART defines a logical CPU, i.e., a concurrent process, which is put into a process queue for later execution. The actual physical CPU the process will execute on is chosen by the multi-tasking system software and can’t be specified by
the programmer. This has the advantage that multi-tasking programs written for the Cray X-MP do not have to be aware of the number of CPUs available or how busy they are. The corresponding join operation is accomplished by a call to the TSKWAIT system routine

\[
\text{CALL TSKWAIT}(\text{taskid})
\]

which makes the parent program wait until the task identified by taskid has terminated. Therefore, TSKWAIT functions as a synchronization point for the several tasks.

Below is an example multi-tasking program in CFT FORTRAN to spread a vector add over two CPUs:

```fortran
PROGRAM ADD
COMMON/GLOBAL/ A(1000), B(1000), C(1000)
INTEGER TASKID(2)
EXTERNAL ADDHALF
DATA B/1000*1.0/, C/1000*2.0/
TASKID(1) = 2
N = 1000
C start up child task to compute upper half of vector
CALL TSKSTART(TASKID, ADDHALF, N/2 + 1, N)
C have parent compute lower half of vector add
CALL ADDHALF(1, N/2)
C have the parent wait for child task to terminate
CALL TSKWAIT(TASKID)
C ... other stuff
STOP
END

SUBROUTINE ADDHALF(N1, N2)
COMMON/GLOBAL/ A(10000), B(1000), C(1000)
DO 10 I = N1, N2
   A(I) = B(I) + C(I)
10 CONTINUE
RETURN
END
```

Here the two tasks -- the one initiated by TSKSTART and the main program -- are independent and no intertask communication is required. Since the Cray X-MP is a shared memory machine, intertask communication is achieved by sharing variables by way of a global COMMON area. The multi-tasking software provides locks (semaphores) for critical regions of code, specifically, the use of these variables shared by several tasks. For example, to update the shared variable `COUNT`, each task would surround the update with a call to `LOCKON` and `LOCKOFF` as in the following:

```fortran
CALL LOCKON(LOCK3)
COUNT = COUNT + 1
CALL LOCKOFF(LOCK3)
```

where `LOCK3` is an integer variable assigned to and identifying the lock.

Therefore, to utilize multi-CPU's on the Cray X-MP, the programmer must manually partition the code into subroutines which are forked off as concurrent tasks. The programmer is responsible for identifying any commonly shared variables used by the tasks and protecting those sections of code which use the shared variables with locks. Also, the programmer is responsible for balancing the amount of computation between concurrent tasks to minimize the idle time of CPUs.
Multi-tasking on the Cray X-MP has some costs. First, the calls to the TSKSTART, TSKWAIT, LOCKON and LOCKOFF routines take time and there is system overhead associated with the multi-tasking system to coordinate and schedule the tasks. However, for some applications, multi-tasking is effective in gaining higher performance.
Chapter 3 Exercises

1. Discuss the differences between a register-to-register architecture and a memory-to-memory architecture.

2. In the context of this chapter, what is strip mining?

3. The ABC computer has a pipelined CPU which can produce a floating point add and a multiply every clock period (8 nanoseconds). What memory bandwidth is needed for a typical arithmetic operation?

4. For the computer in problem 3, assume you have available lots of 80 nanosecond memory. Design a memory system which can achieve the memory bandwidth you computed in problem 3.

5. Vector computers usually have the ability to mask selected elements in a vector. Why is such masking needed in a vector computer?

6. Given vector $A = \{1.0, 2.0, 3.0, 4.0, 5.0\}$ and masking vector $M = \{10010\}$. What is the result of the compress instruction using $A$ and $M$ as arguments?

7. What is the stride of the following FORTRAN code?

```fortran
REAL X(3,7)

DO 10 I = 1, 7
   X(2, I) = 4.0
10 CONTINUE
```

8. Why is stride important on vector computers?

9. What is the vector length used by an arithmetic pipeline in the following code? How could you improve the vector length?

```fortran
DO 30 K1 = 1, 64
   DO 50 K2 = 10, 20
      C(K1 + 7, K2) = 5.0 * D(K1, K2)
50 CONTINUE
30 CONTINUE
```

10. If your application had mostly vectors of 50 elements long, would you execute the program on a Cray-1 or a CDC 205. Justify your answer.

11. In some sort algorithms, the array values needed to be moved (permuted) correspond to the bit-reversal (i.e., flip the binary value horizontally) of the index. For example, if the index is 4 (100 in binary), move the value to 1 (001 in binary) and the value at 1 (001 in binary) is moved to 4. Show how to use the scatter and/or gather operations to generate this bit reversal. Give an example using a 16 element array.

12. On the CDC 205, a linked triadic operation has double the performance compared to a simple two argument vector operation. Why? Also, why is $n^{1/2}$ almost the same for the two cases?