Computer Performance: A Tutorial

Lloyd D. Fosdick
Carolyn J. C. Schauble
Elizabeth R. Jessup

December 19, 1994

DRAFT: December 19, 1994

High Performance Scientific Computing
University of Colorado at Boulder

The following are members of
the HPSC Group of the Department of Computer Science
at the University of Colorado at Boulder:

Lloyd D. Fosdick
Elizabeth R. Jessup
Carolyn J. C. Schauble
Gitta O. Domik

Contents

Computer Performance: A Tutorial*

Lloyd D. Fosdick
Carolyn J. C. Schauble
Elizabeth R. Jessup

December 19, 1994

[DRAFT: December 19, 1994]

1 Introduction and Background

Many types of computers currently exist including personal computers, workstations, mainframes, and supercomputers. Some perform well for word-processing; some have excellent graphical capabilities; and some are designed for efficient number-crunching. Given an application, how does one decide which computer to use or to purchase? How can one determine the efficiency of a program running on a certain architecture? How can one judge the performance of a computer?

The major reason for measuring the performance of a computing system is for predicting its performance on some computations you wish to do. But why do you want to predict performance? Perhaps you are interested in purchasing a system. Before investing the money, you would be wise to study the performance of the system. Of course vendors usually supply you with

*This work has been supported by the National Science Foundation under an Educational Infrastructure grant, CDA-9017953. It has been produced by the HPSC Group, Department of Computer Science, University of Colorado, Boulder, CO 80309. Please direct comments or queries to Elizabeth Jessup at this address or e-mail jessup@cs.colorado.edu.

plenty of performance data, but it is best if you supplement this information with your own observations. Not only are you less likely to be biased, but if you understand how to do these kinds of measurements, you are likely to come up with a more accurate measure *for the way in which you intend to use the system.* The latter point is important because the speed of high-performance computing systems tends to be very dependent on the nature of the jobs run on them. The system may perform very well on one class of problems and be a complete bust on another.

Another reason why you would want to predict performance is in striving for efficiency in a computation. From measurements of arithmetic speeds on simple problems, you can get some idea of the expected performance on the problem at hand. If the actual performance on your problem falls short of what you expect, then you want to understand why, and you probably take steps to improve the performance. Not only does this save you money, it is likely to expose mistakes that you made, lead you to a better algorithm and, in the end, allow you to do more work better.

Predicting computer performance is difficult. It is more of an art than a science, and here we want to begin learning that art. The reason why it is difficult is that many factors beyond the basic operation speeds influence the performance. One can measure the time it takes to add, multiply, and divide numbers — though even this is rather tricky — but using these measurements to predict the performance of a major computation like simulating the flow of fluid in a pipe is almost impossible. At best you can predict a bound on the performance, that is, the best performance you can hope to achieve. What you will actually achieve is another matter. To understand why this is so, and to learn how to deal with it, you need to understand the many factors that can influence performance. This tutorial provides a starting point for this.

Section 2 of this tutorial begins by defining computer performance and by presenting methods of measuring CPU speed and computer performance. Section 3 covers benchmarking, including a discussion of several of the commonly used benchmarks. Section 4 explains the effect of compiler optimizations on performance results while section 5 talks about the effect of some architectural factors on performance. Finally, section 6 introduces additional performance issues related to vector and parallel computing. In particular, this tutorial is directed toward the study of computer performance as related to high-performance computers in a UNIX environment.

2 Computer Performance

By *computer performance*, we mean the effectiveness of the operation of a computer, usually on a given application or benchmark. This notion of performance includes speed, cost, and efficiency.

A number of factors affect the performance of a computer:

- CPU execution time for
  - — register operations
  - — integer operations
  - — floating-point operations
  - — string operations
- Memory access time to read and write data
  - — in cache
  - — in main memory
  - — in auxiliary memory
- Memory size
- Systems and file service
- Compiler
- Input/Output
- Communication, especially with regard to parallel computers

In this tutorial, we are concerned mostly with the first of these: CPU performance or speed, in particular, as it relates to the speed of floating-point operations. High-performance scientific computing implies creating numerically-intensive programs and running them on those computers best able to execute them efficiently.

2.1 Determining the speed of a CPU

The CPU speed of a computer is related to the performance of that machine, but it is not the same thing. The CPU speed tells us how fast the machinery of the CPU works; the performance metrics tell us how fast the CPU can execute given tasks.

Recall the following definitions:

- **millisecond (msec):** one-thousandth (0.001 or 1×10^{-3}) of a second.
- **microsecond (μsec):** one-millionth (0.000001 or 1×10^{-6}) of a second.
- **nanosecond (nsec):** one-billionth (0.000000001 or 1×10^{-9}) of a second.
- **Hertz (Hz):** one cycle per second; an international unit for frequency.
- **MegaHertz (MHz):** one million cycles per second.

A CPU cycle is the finest unit of operation we consider within the CPU. The speed of the CPU clock may be defined as follows:

- **Clock rate:** the number of clock cycles executed per second (usually in units of MHz)
- **Clock speed:** the length of time it takes to execute a single CPU clock cycle (usually in units of microseconds or nanoseconds).

These two definitions are the reciprocals of each other. If the rate of a CPU clock is 50 MHz, then the length of a single clock cycle for that CPU is 20 nanoseconds. And if one clock cycle occurs every 12.5 nanoseconds, then the clock rate is 80 MHz.

Table 2 on page 28 includes the clock rate of some current workstations. The processor cycle times of a few Cray models are shown in table 3 on page 52 while the cycle times of several parallel computers are given in table 6 on page 61. Notice that the range of processor speeds for the workstations is not that different from the large supercomputers.

In order to determine the performance of a computer, we need to be able to time its execution on various tasks. The following terms are used to describe different types of times related to computer execution.

- **user time:** The amount of time the CPU is active for a given process, excluding operating system overhead.
- **system time:** The amount of time the operating system is active for the given process.
- **elapsed time:** The difference between the time when the timed process begins on the given processor and the time that it ends. This is the total time the process required to run. Since most computers allow multiprocessing, the *elapsed time* for a process is usually more than the sum of the *user time* and the *system time*.
- **wall clock time:** Another term for *elapsed time*.
- **response time:** Still another term for *elapsed time*, since this is the length of time the user waits for the execution of his program to complete.

One additional term related to computer execution time is **throughput**: the amount of work (jobs or operations) that the computer can complete in a given amount of time.

Since each machine is different and timing utilities may behave differently on different machines, timing the execution of programs can become a difficult procedure. This section describes some tools available for this process.

In *UNIX* environments, there are many tools for measuring time. The *UNIX* command, `man -k time`, produces a long list. The following utility programs can be used to assist in timing program executions. There are advantages and disadvantages with each.

UNIX `time` **command:** In both the Bourne shell (*sh*) and the C shell (*csh*), the *UNIX* `time` command times the execution of another command, producing the elapsed, user, and system times upon the completion of the command. For instance, in the Bourne shell,

```
time latex perf
```

compiles the LaTeX file named `perf.tex` and then reports on the time this process took:

```
11.7 real          5.3 user          0.3 sys
```

This information shows that the total user time for this operation is 5.3 seconds while the system time is only 0.3 seconds. The elapsed time is 11.7 seconds. Other executions of the same command may produce slightly different results.

In the C shell, the result of the `time` command is in a different format. It also provides additional information. For example, the command

```
time latex perf
```

executes the `latex perf` command and then prints a line showing the time of execution:

```
5.3u 0.5s 0:31 18% 0+1124k 97+18io 129pf+0w
```

Reading this line from left to right, the figures should be interpreted in the following manner. The first notation, `5.3u`, means that the total user time is 5.3 seconds while the second figure tells us the system time is only 0.5 seconds. The elapsed time for this operation is 31 seconds. In this example, the percentage of user time to elapsed time is 18% (5.3/31). The remaining figures provide information on input, output, and storage requirements for the program execution.

The user time is the same in both cases since the same command is being executed under both shells. But notice the difference in the two elapsed times: 11.7 seconds versus 31 seconds. Three additional executions of the `time latex perf` command in the Bourne shell resulted in elapsed times varying from 6.4 seconds to 10.7 seconds, while three more executions of the `time latex perf` command under C shell produced elapsed times from 8 seconds to 12 seconds. This difference in the elapsed times may be caused by changes in the load of the machine executing the command. Also notice that the sum of the user time and the system time is usually less than the elapsed time. These results suggest the dubious value of using elapsed time for careful performance measurements, particularly in a multiprogramming environment.

The main problem with the `time` command in either shell is that the resultant times are only correct to one-tenth of a second. Since most current CPU cycle times are in nanoseconds, millions of cycles must be executed to produce a non-zero time.

Another problem is that the timing is done outside the application being timed. In other words, the `time` command first records the current time, and then it starts the execution of the application. When the application execution has completed, the final time is recorded, and the differences are displayed in the appropriate format. This does not permit the timing of a small set of instructions within a program or the timing of an inner loop of a program; only the execution of entire programs can be timed with this command.

***UNIX* `date` command:** The *UNIX* `date` command returns the current date and time (correct to the current second) when entered at the *UNIX* prompt. It can also be called within a *C* program using the `system` function. By employing string functions to separate out the values for the hours, minutes, and seconds from the returned string expression, it is possible for this command to record times during the execution of a program and thus to time inner portions of the program.

While this utility has the advantage of providing time values during the execution of a program, it has the disadvantage of only being correct to seconds. Again, this means that tens to hundreds of millions of cycles must be executed to produce a non-zero time.

Most *UNIX Fortran* compilers provide a similar function.

The *Fortran* programming language has traditionally been used for high-performance scientific computing. As a result, most *Fortran* compilers provide additional timing routines to assist the scientific and performance programmer:

`ETIME`: There exists an intrinsic *Fortran* procedure named `ETIME` in many *Fortran* compilers; a double-precision version of this is named `DTIME`. These procedures return the time that a process has used since it started. Thus it is as if a private clock, private to a given process, is started when that process starts; it runs only while that process is running, stopping temporarily when the process is idle and resuming when the process becomes active. Thus it is not the same as wall-clock time because it excludes the effects of multiprogramming. In fact, `ETIME`, gives the two components of the time used by the process, user time and system time.

For a DECstation 5000/240, the single-precision time given by `ETIME` is in microseconds; in section 2.2, we show how to determine that the `ETIME` clock is advanced once every 4 milliseconds. This is a finer clock than either the *UNIX* `time` or `date` commands and allows the timing of smaller sets of instructions.

Warning: Different *Fortran* compilers define the parameters for `ETIME` differently. Also, different units of time may be measured on different machines.

`SECOND`: Some *Fortran* compilers, including the Cray compiler, contain a function named `SECOND` that provides the total time used by the calling process. Like `ETIME`, this function returns a finer unit of time. On the Cray Y-MP, the `SECOND` timer clicks once for every processor clock cycle; in section 2.2 we determine that it ticks once every 6 nanoseconds. Timing done with this function is even more accurate than the others discussed above.

`SYSTEM_CLOCK`: The `SYSTEM_CLOCK` function returns the total time allocated to a process. It is included in both *Fortran-90* and *High Performance Fortran* (*HPF*), and it returns integer data from the real time clock of the processor on which the program is executing.

The three optional output parameters for this function are `COUNT`, `COUNT_RATE`, and `COUNT_MAX`. The first parameter, `COUNT`, provides the current value of the processor clock; it ranges in value from 0 to `COUNT_MAX`. The parameter, `COUNT_MAX`, is the maximum integer value that can be held in the real time clock. The parameter, `COUNT_RATE`, returns the number of processor clock counts per second to provide the *granularity* of the processor clock; this may be in thousandths or hundredths meaning that the clock is accurate to milliseconds or tens of milliseconds. Both `COUNT_MAX` and `COUNT_RATE` are dependent on the real time clock used by the given processor.

The *granularity* of a clock relates to the size of the smallest interval

```
* mytime.f
      SUBROUTINE MYTIME(TIME)
* Time for DEC 5000/240, values are in seconds.
      EXTERNAL ETIME
      REAL T(2),TIME
      TIME =  ETIME(T)
      RETURN
      END
```

Figure 1: A *Fortran* timing routine for the DEC 5000/240 that uses `ETIME`.

measured by the clock; it is the time between instants when the clock time is advanced one unit. A fine granularity implies a very small interval (perhaps in nanoseconds) while a coarse granularity implies a larger interval of time (such as seconds or tenths of seconds).

2.2 Determining Clock Granularity

It is important to know the granularity of the clock for a timing routine. The finer the clock granularity, the more accurate the information obtained by the routine. A coarse clock granularity requires longer sequences of instructions to be timed in order for them to be measured by the clock correctly.

Given a timing routine, such as `ETIME`, on a certain machine, how does one determine the granularity of the clock used by that routine? This can be an intricate procedure, dependent on the architecture and the routine. We provide two examples in this section.

2.2.1 `ETIME` on the DEC 5000/240

First we create a timing routine called `MYTIME` that uses `ETIME`, as is shown in figure 1. The value assigned to `TIME` is the user time plus system time elapsed since the start of execution of the program. We do not use the values `T(1)` and `T(2)`, that provide user time and system time, respectively.

To determine the precision of the value returned by `MYTIME` on the DEC 5000/240, we run an experiment in which we time a program segment that executes in an interval somewhat shorter than the granularity of the clock. The program used is given in figure 2. The subroutine, `MY_DATE_TIME`, also called by this program merely provides a date and time stamp for the run of the program.

```
* timetst.f:  Test clock granularity
      IMPLICIT NONE
      INTEGER J,K,SUM
*****************************************************************
* J,K - loop indices.
* SUM - variable to accumulate sum.   (OUT)
*****************************************************************
      REAL T1, T2
*****************************************************************
* T1,T2 - time readings.  (difference OUT)
*****************************************************************
      CHARACTER*10  DATEBUF, TIMEBUF
      CHARACTER*40  PROGRAM_LABEL
      PARAMETER(PROGRAM_LABEL='timetst, 28 august 1992, ldf/cjcs')
*****************************************************************
* DATEBUF - Current date.  (OUT)
* TIMEBUF - Current time.  (OUT)
* PROGRAM_LABEL - Name, creation date, author.  (OUT)
*****************************************************************
      EXTERNAL  MY_DATE_TIME, MYTIME
*****************************************************************
* MY_DATE_TIME - a function returning current date/time.
* MYTIME - a function returning time.
*****************************************************************
* Write header.
      WRITE(*,*) PROGRAM_LABEL
      DATEBUF = '          '
      TIMEBUF = '          '
      CALL  MY_DATE_TIME(DATEBUF, TIMEBUF)
      WRITE(*,*) 'RUN DATE_TIME  ', TIMEBUF, '  ', DATEBUF
      WRITE(*,*)
      WRITE(*,*) 'TIME(sec), SUM'
      WRITE(*,*)

* Main loop to time 20 samples of 10000 additions.
      DO 20 K = 1,20
         CALL MYTIME(T1)
         SUM = 0
         DO 10 J = 1,10000
            SUM = SUM + 1
  10     CONTINUE
         CALL MYTIME(T2)
         WRITE(*,999) T2-T1, SUM
  20  CONTINUE

* Done.
      STOP
 999  FORMAT(1X,1PE12.5,1X,I7)
      END
```

Figure 2: A *Fortran* program to test the granularity of the `ETIME` clock routine for the DEC 5000/240.

```
timetst, 28 august 1992, ldf/cjcs
RUN DATE_TIME  14:10:31    01-Sep-92

TIME(sec), SUM

 0.00000E+00   10000
 0.00000E+00   10000
 0.00000E+00   10000
 3.90600E-03   10000
 0.00000E+00   10000
 0.00000E+00   10000
 0.00000E+00   10000
 0.00000E+00   10000
 0.00000E+00   10000
 0.00000E+00   10000
 3.90600E-03   10000
 0.00000E+00   10000
 0.00000E+00   10000
 0.00000E+00   10000
 0.00000E+00   10000
 0.00000E+00   10000
 0.00000E+00   10000
 3.90600E-03   10000
 0.00000E+00   10000
 0.00000E+00   10000
```

Figure 3: Output from `timetst.f` as run on the DEC 5000/240.

This program makes 20 independent measurements of the time to execute the loop in which the sum

$$1 + 1 + 1 + \ldots + 1$$

is computed. The output from this program is shown in figure 3.

From these results, you can see that the granularity is 0.003906 seconds. You can confirm this by increasing the number of iterations of the loop from 10000 to a higher value. As you gradually increase, you start to see times of 0.007812, 0.023436, and so on. Note that $0.003906 \approx 2^{-8}$, $0.007812 \approx 2 \times 2^{-8}$, etc. The second column is simply a sanity check: it confirms that

```
* mytime_cray.f
* Time for Cray Y-MP, values are in seconds.
***************************************************************
        SUBROUTINE MYTIME(T)
        REAL T
        EXTERNAL SECOND
***************************************************************
* T - the total elapsed time.
* SECOND - the Cray Fortran routine for returning elapsed time.
***************************************************************
        T = SECOND()

        RETURN
        END
```

Figure 4: A *Fortran* timing routine for the Cray Y-MP that uses `SECOND`.

the inner loop is executed 10000 times.

When you run this experiment several times, you do not get identical results; the locations of the nonzero values change because the *ticks* of the clock are initiated by an independent process. Thus, if your process starts immediately before a clock tick you find a nonzero value early in the list, but if your process starts shortly after a clock tick your first nonzero value appears later.

These results show that the granularity of the clock on the DECstation 5000/240 is about 4 msec. To measure the time of execution of a block of code, the clock is read immediately before entering the block and again when leaving the block. The difference between the times is the time spent in the block, and it should be evident that error in this measurement is less than 8 msec. Thus, if we time a block of code in this way and find that the execution time is 1 sec., the relative error in this measurement is less than 0.8%.

2.2.2 `SECOND` on the Cray Y-MP

The function, `SECOND`, is a timing tool that is used with *Fortran* on the Cray. A revised version of the MYTIME function for the Cray is shown in figure 4.

As with the `ETIME` function used by the DECstation timing routine, the next step is to determine the granularity of the clock used by the Cray `SECOND` function. If you try to use the DECstation version of the program, `timetst.f`, above, you may not get the same kind of results. The granularity of the `SECOND` timer on the Cray is much smaller than the granularity of the

`ETIME` routine on the DECstation; in fact, more than one tick of the Cray `SECOND` timer occurs during a single integer add on the Cray.

Consider the program, `timetst_cray.f`, shown in figure 5. Since the Cray Y-MP is a vector processor (see section 6 for more about vector processors), the compiler tries to optimize the code by including vectorizations (see section 4 for more on compiler optimizations). We need to make sure that the code being timed is not vectorized. For this reason, the compiler directive, `CDIR$ NOVECTOR`, is placed just before the loops containing the scalar operations to be timed; this tells the compiler not to perform any vectorizations on this code. To avoid having the compiler remove the loop as part of the optimization, we have placed a `WRITE` statement inside the loop; this forces the loop to be executed.

The output of this program is shown in figure 6. Observe that none of the times shown in the output are zero. Instead they are all in the neighborhood of 2.7 to 2.9 microseconds. However, consider the differences between the times; these are all multiples of 6 nanoseconds. Thus, the clock granularity for the Cray `SECOND` function must divide 6 nsec.; that is, it could be 6 nsec., 3 nsec., 2 nsec., etc. As shown in table 3 on page 52, the cycle time of the Cray Y-MP is 6 nsec.

Comment: the times given in this output are much larger than an integer add time should be. This is because only a single add is timed, and that time includes the overhead of calling and returning from the Cray `SECOND` function. If we wanted a more accurate timing of the integer add operation – instead of trying to determine the clock granularity, we should include enough repetitions of the operations to be timed to make the calling overhead insignificant. We discuss this further in section 2.4.

2.3 Measures of computer performance

The performance of one computer is better than the performance of another computer when the same application program executes faster on the first computer than on the second. In other words, computer performance is measured by the reciprocal of execution speed. However, many factors besides the CPU speed of a machine affect the execution speed of any given application program. These factors may produce conflicting performance results when a machine is tested using multiple applications.

```
* timetst_cray.f:  Test clock granularity
      IMPLICIT  NONE
      INTEGER K,SUM
*********************************************************************
* K - Loop index.
* SUM - Accumulating sum of timed additions.   (OUT)
*********************************************************************
      REAL T1, T2, SECOND
      EXTERNAL SECOND
*********************************************************************
* T1, T2 - Beginning and ending times for each add operation.
*       (difference OUT)
* SECOND - Cray timing function.
*********************************************************************
      CHARACTER*10  DATEBUF, TIMEBUF
      CHARACTER*40  PROGRAM_LABEL
      PARAMETER (PROGRAM_LABEL = 'timetst_cray, 7 sept 92, ldf/cjcs')
*********************************************************************
* DATEBUF - Current date.    (OUT)
* TIMEBUF - Current time.    (OUT)
* PROGRAM_LABEL - Name, creation date, author.   (OUT)
*********************************************************************
* Write header.
      WRITE (*,*) PROGRAM_LABEL
      DATEBUF = '          '
      TIMEBUF = '          '
      CALL MY_DATE_TIME(DATEBUF, TIMEBUF)
      WRITE (*,*) 'RUN DATE_TIME  ', TIMEBUF, '  ', DATEBUF
      WRITE (*,*)
      WRITE (*,*) 'TIME(sec), SUM'
      WRITE (*,*)

* Timed loop of 20 adds.
CDIR$ NOVECTOR
      SUM = 0
      DO 20 K = 1,20
          T1 = SECOND()
          SUM = SUM + 1
          T2 = SECOND()
          WRITE(*,999) T2-T1, SUM
  20  CONTINUE

* Done
      STOP
 999  FORMAT(1X,1PE12.5,1X,I7)
      END
```

Figure 5: A *Fortran* program to test the granularity of the `SECOND` clock routine for the Cray Y-MP.

```
timetst_cray, 7 sept 92, ldf/cjcs
RUN DATE_TIME  15:30:40    09/07/92

TIME(sec), SUM

 2.90400E-06       1
 2.82600E-06       2
 2.79600E-06       3
 2.79600E-06       4
 2.79600E-06       5
 2.79600E-06       6
 2.79600E-06       7
 2.79600E-06       8
 2.83800E-06       9
 2.80800E-06      10
 2.82600E-06      11
 2.79600E-06      12
 2.79600E-06      13
 2.79600E-06      14
 2.79600E-06      15
 2.82600E-06      16
 2.79600E-06      17
 2.79600E-06      18
 2.79600E-06      19
 2.81400E-06      20
```

Figure 6: Output from `timetst_cray.f`.

The following units have been developed over the years to describe measurements of performance:

- **mips:** Originally, this term referred to the speed of a computer in terms of the number of `m`illions of `i`nstructions `p`er `s`econd it could perform. For example, if a set of 15,000 instructions runs in 3 milliseconds on a certain machine, then the mips for that application running on that machines is 5; that is,

$$\begin{aligned} \frac{15000\ instrs}{3\ millisecs} &= \frac{15000000\ instrs}{3\ secs} \\ &= \frac{5000000\ instrs}{1\ sec} \\ &= 5\ mips \end{aligned}$$

  The reader should note that the mips measurement is highly dependent on the instruction mix used to produce the execution times. A program executing 100,000 instructions with a high percentage of simple integer operations runs in a shorter amount of time than a program executing the same number of instructions with a high percentage of complex instructions, including floating-point operations. Hence, two different programs with the same number of instructions can produce different mips measurements on the same machine.

  Another problem with the accuracy of the mips performance measurement is that some architectures have a simple instruction set and others are designed with a more complex set of instructions. While programs running on a simple instruction set architecture may compile into more instructions than the same program on the more complex instruction set architecture, simpler instructions usually execute more quickly than complex instructions. Thus, it can be difficult to compare the mips of two different machines even when using the same program.

  The term, mips, is still used in the manner described above. However, it is also used to describe the results of the MIPS[1] benchmark (see

[1] MIPS also refers to a family of RISC architectures developed by MIPS Computer Systems, Inc., a subsidiary of Silicon Graphics, Inc. This family includes the MIPS R2000, MIPS R3000, MIPS R4000, and MIPS R10000 microprocessor chips.

section 3). In this context, the millions of instructions per second are correlated to the performance of a VAX 11/780.

- **mops:** This term defines the speed of a computer in terms of the number of `m`illions of `op`erands `p`er `s`econd it can handle. These operands may refer to register values or to data located in the cache or in the memory of the machine. The machine instructions for some processors contain only one operand per instruction; in this case, the mops for the processor should be the same as the mips. Other processors may have two, three, or more operands in an instruction; the mops for these machines are greater than the mips.

  Consider the above example where a set of 15,000 instructions runs in 3 milliseconds on a certain machine. Assume that each instruction for that machine contains three operands. Then the mops is 15 for this application; that is,

$$\begin{aligned} \frac{15000\ instrs\ \times\ 3\ operands/instr}{3\ millisecs} &= \frac{45000000\ operands}{3\ secs} \\ &= 15\ mops \end{aligned}$$

- **CPI:** This term is used to define the speed of a computer in terms of the number of clock cycles per instruction.

  Continuing with the above example, assume that the clock rate of the given computer is 20 MHz. This means that each clock cycle takes 50 nanoseconds. The earlier examples have shown that this machine executes at 5 mips; that is, it executes five million instructions per second or one instruction per 200 nanoseconds. Hence each instruction requires an average of 4 clock cycles, and the machine operates at 4 CPI on average.

- **flop:** This common performance unit defines the speed of a computer in terms of the number of `fl`oating-point `op`erations it can perform in one second.

  Returning to the same example, if we assume that each instruction performs one floating-point operation, then the number of floating-point operations is the same as the number of instructions, and the machine operates at 5,000,000 flops.

However, it is highly unlikely that all the instructions are floating-point operations; for example, some may be integer operations that control loops or perform subscript calculations. If we instead assume that only four-fifths of the instructions are floating-point operations, then the machine operates at 4,000,000 flops; that is,

$$\begin{aligned}\frac{4}{5}\, floating\!-\!point\; operations/instr \quad & \times\; 5,000,000\; instrs/sec \\ & =\; 4,000,000\; flops.\end{aligned}$$

The main problem with this performance measurement unit as we have defined it is that not all floating-point operations require the same number of clock cycles. For instance, a floating-point division may take 4 to 20 times as many cycles as a floating-point addition. Thus, the flop must be more precisely defined any time it is used to measure machine performance. In some contexts, all floating-point operations are included in the count of flops. In others, only the multiplications and additions are included. (The LINPACK benchmark uses this definition of the flop.) Yet another definition of flop proposed by C.B. Moler [Golub & Van Loan 83] includes some time for access of array elements in addition to multiplications and additions. This flop is defined by

$$s = s + a(i,k) * b(k,j). \tag{1}$$

All of these definitions are valid ones, but you must be careful to state which one you are using.

The flop is an especially popular measure of machine performance on scientific and other mathematical programs, but it is not a reasonable measure for programs or benchmarks using few or no floating-point operations.

- **megaflop (Mflop):** A speed of a computer as the number of `M`illions of `fl`oating-point `op`erations it can perform in one second.

  For instance, the machine in the example above operates at 4 Mflops.

- **gigaflop (Gflop):** Another multiple of the *flop*, this term defines the speed of a computer as the number of billions of floating-point operations it can perform in one second. As displayed in table 1, most of today's supercomputers are capable of executing in Gflops.

- **teraflop (Tflop):** Still another multiple of the *flop*, this term defines the speed of a computer as the number of trillions of floating-point operations it can perform in one second. This is the rate that most new supercomputer designs are attempting to achieve.

One other interesting term used to describe computer performance is the *peak performance* of a machine. This is simply the maximum number of flops that a machine can theoretically obtain. It is almost never achieved. According to Dongarra [Dongarra 94],

> The theoretical peak performance is determined by counting the number of floating-point additions and multiplications (in full precision) that can be completed during a period of time, usually the cycle time of the machine.

Full precision in this context denotes double-precision arithmetic that uses 64 bits on most architectures. However, some machines do not follow the IEEE arithmetic standard. For instance, the Cray Y-MP uses 64 bits to represent a single-precision value; in this case, the single-precision results would be used for the *theoretical peak performance* (TPP).

Table 1 provides a short list of supercomputers, with the name of the machine series, the name of the manufacturer, and the TPP given in Mflops. The figures are taken from [van der Steen 93].

Still other units of computer performance measurement refer to specific benchmarks programs. Such units include *SPECmarks* and the other definition of *mips*. These are discussed in the next section.

2.4 Timing elementary operations

In this section we discuss how to time the elementary arithmetic operations on the DECstation 5000/240. This system uses a MIPS R3000 CPU and R3010 FPU chipset [Kane 88, DEC 91], operating at 40 MHz (25 nsec cycle time). Thus we can expect that arithmetic operations require some small

Machine	Manufacturer	Number of Processors	TPP (Mflops)
CM-2	Thinking Machines	2,048	20,000
CM-5	Thinking Machines	16,384	2,000,000
Cray 2	Cray Research	8	4,000
Cray Y-MP	Cray Research	16	15,000
Cray T3D	Cray Research	2,048	307,000
Cray-3	Cray Computer Corp	16	16,000
IBM ES/9000	IBM	6	2,700
IBM 9076 SP1	IBM	64	8,000
Intel iPSC/2	Intel	128	250
Intel iPSC/860	Intel	128	7,700
Intel Delta	Intel	512	20,000
Intel Paragon	Intel	4,000	300,000
KSR1	Kendall Square Research	32	1,300
KSR2	Kendall Square Research	5000	400,000
MP-1	MasPar	16,384	550
MP-2	MasPar	16,384	2,400
NEC SX-A	Nippon Electric Company	4	22,000

Table 1: A list of selected supercomputers with their manufacturer's name and their theoretical peak performances in megaflops and their maximum number of processors.

multiple of 25 nsec. Although we describe this timing procedure for the DEcstation 5000/240, the basic concepts apply to the timing of any very fast operation on any machine.

As we observed previously, the granularity of the clock we use to measure time on the DECstation is only $\frac{1}{256}$ seconds, or about 4 msec. So how do we measure an arithmetic operation time that we expect to be much smaller than the granularity of the clock? The basic idea is simple: we measure the time, T, it takes to do a large number, N, of the operations; then the operation time is given by T/N. The details are best explained with the outline of the code segment for the timing shown in figure 7.

The statement `CALL MYTIME(T1)` is used to get an initial reading of the

```
      CALL MYTIME(T1)
      DO 10 I = 1, NREP
*          {put R1 repetitions of "op" here}
   10 CONTINUE
      CALL MYTIME(T2)
      DO 20 I = 1, NREP
*          {put R2 repetitions of "op" here}
   20 CONTINUE
      CALL MYTIME(T3)
      OPTIME = ((T2-T1) - (T3-T2))/(NREP*(R1-R2))
*          OPTIME is the estimated cost of "op" in seconds
```

Figure 7: Outline of timing code segment.

time `T1`. After completing the computation we are measuring, we obtain the new time `T2` with the statement `CALL MYTIME(T2)`. Then the difference between the readings `(T2 - T1)` is the elapsed time. Now this elapsed time includes not only the time to do the N operations, but also the overhead for the loop control arithmetic. However, the latter should not be included in the operation time as it is just an artifact of the technique we use to measure the time.

Removing the loop overhead time is the reason for the second time measurement: the measurement starting with the statement `CALL MYTIME(T2)` and ending with the statement `CALL MYTIME(T3)`. Since the amount of loop control arithmetic is the same for both `DO` loops, the time for the loop control arithmetic is canceled out by the expression `((T2-T1) - (T3-T2))`; this expression contains only the time for performing `NREP*(R1 - R2)` repetitions of "op". Thus the statement

```
OPTIME = ((T2-T1) - (T3-T2))/(NREP*(R1 - R2))
```

gives the time for one execution of "op".

To get correct timings, we need to use values of `R1` and `R2` that are not altered in any way by the optimization stages of the compiler (see section 4). If these values are too small, the optimizer (level `O4`) may unroll the loop and thus modify the amount of loop control arithmetic from what you think it is; if the values are too big, then the code is clumsy with an unnecessarily

```
* add.f
* Double precision add test
            ⋮
      DOUBLE PRECISION  X, Y
            ⋮
      SUMTIME = 0
            ⋮
      DO 130 J = 1,NSAMP
         CALL MYTIME(T1)
         DO 110 I=1,NREP
* 16 copies of timed op X = X+Y follow.
            X = X+Y
              ⋮
            X = X+Y
  110    CONTINUE
         CALL MYTIME(T2)
         DO 120 I = 1,NREP
* 8 copies of timed op X = X+Y follow.
            X = X+Y
              ⋮
            X = X+Y
  120    CONTINUE
         CALL MYTIME(T3)

* Compute elapsed time and update variables.
         DT = (T2-T1)-(T3-T2)
         SUMTIME = SUMTIME + DT
  130 CONTINUE

* Compute and write average time and megaflops.
       AVGTIME = SUMTIME/(NSAMP*NREP*(R1-R2))
       MFLOPS = 1/(AVGTIME*1.0E6)
            ⋮
```

Figure 8: Outline of `add.f`, a program to determine the time required for one double-precision addition on a DECstation 5000/240.

```
add1, 31 august 1992, ldf/cjcs
RUN DATE_TIME  10:32:12    03-Sep-92

X, Y, NREP, NSAMP, DUP
MINTIME(sec), MAXTIME(sec), AVG.OP.TIME(sec), MFLOPS, X

1.00000E-01  1.00000E-01  1200000       20   8
 7.2261E-01  7.3042E-01  7.5475E-08  1.3249E+01  5.7600E+07

1.00000E-01  1.00000E-01  1800000       20   8
 1.0859E+00  1.0937E+00  7.5557E-08  1.3235E+01  8.6400E+07
```

Figure 9: Sample output from `add.f`.

large loop body. After some experimentation, we have found that the values of 16 for `R1` and 8 for `R2` work well for the DECstation.

As an example, we measure the cost of double-precision, floating-point addition, `X = X + Y`. It is useful to report the average operation time and the average rate of executing the statement in megaflops.

Figure 8 shows only the most important parts of the program. Here `R1` and `R2`, defined in the code outline above, are 16 and 8 respectively; the variable, `DUP`, represents the difference between `R2` and `R1`. The variable `DT` represents the elapsed time for $8\times$`NREP` executions of the statement `X = X+Y`. The granularity of the clock causes an error in `DT` that could be as large as 16 msec.; notice how `DT` is computed and remember that each time (`T1, T2, T3`) in the expression on the right-hand side could be in error by almost 4 ms. By observing the minimum and maximum values of `DT` shown in the output, we can verify that the difference between these values is no more than 16 msec. If we find that the difference is larger than this then it is most likely due to an unusually heavy load placed on the system by other users, in which case it would be best to wait for a quieter time to do the measurements.

Figure 9 shows the output from a sample run of this program. The loop parameters have been chosen (`NSAMP` $= 20$ for both runs with `NREP` $= 1,200,000$ and $1,800,000$) so as to yield a value of `DT` in the neighborhood of 1 sec. Given that the measurement error should be less than 16 msec., this choice promises a maximum relative error of about 1.5% in the value reported for the average operation time, represented by the variable AVGTIME.

If you are not careful in constructing the loop body, the compiler can change the loop behind your back when it optimizes the code! For example, if you use a sequence of statements in the loop like

```
X = Y + Z
X = Y + Z
X = Y + Z
  ...
```

then the compiler can recognize that the first two statements are ineffective. Indeed the whole loop is ineffective and the compiler may remove it; whether or not it does this depends on the compiler and the level of optimization. To prevent this from happening, we use a sequence of statements of the form

```
X = X + Y
X = X + Y
X = X + Y
  ...
```

All of these statements are effective since they contain a recurrence; that is, each statement recomputes a new value for X using the old value. Of course a *really* smart compiler could figure out that 3NSAMP*NREP*DUP*Y is the final value of X and eliminate the loops entirely, but fortunately for us the current compiler is not so smart. Note, by the way, that the final value of X is printed in the rightmost column of the output as a sanity check.

3 Benchmarks

A computer benchmark measures the performance of a computer or a subsystem of a computer by recording the execution time of a program or set of programs for a well-defined task or set of tasks on that computer.

3.1 Purpose of Benchmarks

A benchmark is a set of programs, a single program, or a program segment that is used to measure performance. The time required for a computer to execute the benchmark provides a performance measure.

Benchmarks can be used to compare the performance of two or more machines. After all, when determining that computer to use or purchase for a given application, it is the performance of that computer on the given application that is important. If users are unable to test their applications on different machines, they should use the results of benchmark programs that resemble their applications run on those machines.

3.2 Types of Benchmarks

Benchmarks are written for many purposes: to check the CPU performance of a computer, the file server performance, the input/output interface, the network, the communication speed, etc. Some benchmarks are self-written; that is, they are written by the user needing the information and are internal to the user's organization. However, the code for many commonly-used benchmarks is available over the internet; most of these codes have been modified and fine-tuned over the years and can be used *with care* to produce performance data that can be compared with existing libraries of numbers corresponding to the performance of other machines run with the same benchmark programs. Still other benchmarks belong to special organizations that test new computers and publish the results; such results are considered less biased than those produced by the computer manufacturers.

If a benchmark is taken from existing code, it is termed a *kernel* benchmark. In other words, the benchmark is created from the kernel of an application program, an essential part of the program, such as the main loop or main subloop in the code. A kernel benchmark may only be a loop from the original application or a stripped-down, simplified version of the original code; nevertheless it is representative of the application program. Other benchmark programs are *real* application programs that may be used as benchmark programs as well.

When the code for a benchmark is written specifically to represent a statistical sample of operations or program statements, it is called a *synthetic* benchmark, as opposed to a real or a kernel benchmark. The code for a synthetic benchmark exists for no other purpose.

Some benchmarks may be classified as *novelty* or *toy* benchmarks. These small programs are based on algorithms that do not represent normal application programs. Game or puzzle-solving programs are included in this category of benchmarks, as well as sorting routines.

3.3 Internet Sites for Benchmarks

There is an electronic storage facility for many common benchmarks, including both the source code and the published results for a number of machines. This is called `netlib`, and it is maintained by the Oak Ridge National Laboratory in Tennessee. Other mathematical software and information is kept at this site as well. This material is duplicated at more than one site and can be accessed in several ways:

electronic mail: A listing of the contents of this library can be obtained by electronic mail to `netlib@netlib.att.com` or `netlib@ornl.gov` with a single line as the message body:

```
send index
```

anonymous ftp: To reach `netlib` by anonymous ftp, type

```
ftp netlib.att.com
```

and change to the directory called `/netlib`.

xnetlib: In an X-windows [2] environment, the command

```
xnetlib
```

brings up an interactive window directly connected to `netlib`.

mosaic: Mosaic is an interface to the world wide web (WWW); the mosaic address

```
http://www.netlib.org
```

is an alternate way to reach `netlib`. Clicking on `The NetLib Repository` provides a listing of the available files. And clicking on `Performance Database Server` provides access to compiled performance information.

[2] X-windows is a trademark of M.I.T.

Copies of this library are located at alternate sites all over the world.

The National Institute for Standards and Technology (NIST) also stores most of the common benchmarks. Again to obtain a listing of those benchmarks available there, send electronic mail to `nistlib@cmr.ncsl.nist.gov` with a single line as the message body:

```
send index
```

They do not keep a library of current benchmark results.

There also is a USENET newsgroup named `comp.benchmarks` that discusses the purpose and value of existing benchmarks, publishes results of many benchmarks, introduces new benchmarks, and often supplies source code for benchmarks. Further information about the benchmarks mentioned here as well as other benchmarks can be found in that newsgroup.

In addition, a number of the publically available benchmarks (and some results) can be found at the following `ftp` sites:

```
sony.com:/pub/benchmarks
ftp.nosc.mil:/pub/aburto
netlib.att.com:/netlib/benchmarks
```

Some `mosaic` sites exist as well.

3.4 Common Benchmarks

Some common benchmarks are described in the following subsections.

Table 2 on page 28 provides a short list of high-performance workstations with the name and clock rate of the machine, followed by the performance rated in SPECmarks for the SPEC benchmark and in Mflops for the LINPACK benchmark. Both of these benchmarks are described in further detail below. The SPECmarks quoted came from various articles posted on `comp.benchmarks` or from manufacturer's brochures; the LINPACK values came from [Dongarra 94].

3.4.1 LINPACK

In the 1970's, a set of *Fortran* subroutines was developed to solve systems of linear equations. This set of routines is called LINPACK [Dongarra et al. 79].

Workstation	**Clock**	**SPECmarks**			**LINPACK**
	MHz	**INT92**	**FP92**	**89**	**Mflops**
DEC 5000/200	25.0	19.5	26.7	18.5	3.7
DEC 5000/240	40.0	27.9	35.8	32.4	5.3
DEC 3000-500 AXP	150.0	84.4	127.7	126.1	30.0
DEC 10000-660 AXP	200.0	106.5	200.4	184.1	43.0
HP 9000/750	66.0	48.1	75.0	77.5	24.0
HP 9000/735	99.0	109.1	167.9	146.8	41.0
IBM RS/6000-350	41.6	35.4	74.2	73.7	19.0
IBM RS/6000-980	62.5	73.3	134.6	126.3	38.0
IBM Power2-990	71.5	126.0	260.4		140.0
SGI Indigo R3000	33.0	22.4	24.2	26.3	4.0
SGI Indigo R4000	50.0	57.6	60.3	70.2	12.0
SGI Crimson	50.0	61.7	63.4	70.4	16.0
SUN Sparc 2	40.0	21.8	22.8	25.0	4.0
SUN Sparc 10/30	36.0	45.2	54.0	57.3	9.3
SUN Sparc 10/40	40.0	50.2	60.2		10.0

Table 2: Performance data based on the LINPACK and SPEC benchmarks for selected workstations. The SPECmarks quoted came from various articles posted on `comp.benchmarks` or from manufacturer's brochures; the LINPACK values came from [Dongarra 94].

Since LINPACK was popular in many areas of scientific computing, it is logical that one of its routines has become the source of a kernel benchmark for comparing the floating-point performance of different machines. The LINPACK benchmark is based on the speed of solving a system of 100 simultaneous linear equations using software from LINPACK in double-precision arithmetic. The results of this benchmark are in Mflops and come from the number of floating-point operations for solving a matrix of size 100×100. Results are sometimes given for single-precision arithmetic as well.

LINPACK was the first linear algebra software package to use the Basic Linear Algebra Subprograms 1 (BLAS1). The BLAS1 is a set of subroutines and functions that perform the common scalar-vector operations such as multiplying a vector by a scalar and vector-vector operations such as the dot product. The BLAS1 are written in Fortran 77, but many computer manufacturers also provide a version of the BLAS1 routines written in the assembly languages specific to their machines. The assembler routines are designed to run as fast as possible. Thus, it is easy to optimize a Fortran 77 program that uses the BLAS1 routines by replacing the compiled Fortran 77 BLAS1 with the fast assembler-coded routines. Many of the reported LINPACK results have been produced using these assembler routines.

While LINPACK is still used for benchmarking purposes, it is no longer recommended for solving linear algebra problems. Released in 1992, LAPACK [Anderson et al. 92] replaces both LINPACK and EISPACK [Smith et al. 76], the collection of routines for eigenvalue problems introduced in 1970. LAPACK makes use of the newest, most accurate algorithms in numerical linear algebra. It also makes efficient use of memory by dividing large matrix operations into smaller block matrix operations. LAPACK is built on the BLAS1, 2, and 3. The BLAS2 and 3 perform matrix-vector and matrix-matrix operations, respectively. LAPACK and all of the BLAS are available via `netlib`.

More information on the LINPACK benchmark, including both code and results, is also available from `netlib`.

3.4.2 Livermore Loops (LFK)

The Livermore Loops, also known as the Livermore Fortran Kernels (LFK), are twenty-four *Fortran* kernels, derived from a group of application programs by Lawrence Livermore National Laboratory. An earlier version included only fourteen kernels. Some of the kernels are as simple as an inner (dot) product or a matrix multiply; others include a search loop from a Monte Carlo routine, a two-dimensional explicit hydrodynamics fragment, and an incomplete Cholesky routine with conjugate gradient. Like the LINPACK benchmark, results are provided in Mflops, but the count of floating-point operations has been normalized to account for the disparity in the execution time of different types of operations. A similar version of the kernels exists in *C*.

More information on the benchmark, including both code and results, is available from `netlib`. Also see [McMahon 86].

3.4.3 Whetstones

Designed to compare the floating-point performance of different computers, this benchmark is a single program, originally written in Algol-60. There is a *Fortran* version as well.

The type and frequency of statements in this program were chosen to represent the type and frequency of statements from numerically-intensive programs at the National Physical Laboratory in England, based on statistics gathered in 1970. These statements are translated to instructions for an imaginary computer called *Whetstone*. Hence, this is a synthetic benchmark.

These benchmarks are publically available by ftp at

`netlib.att.com:/netlib/benchmark/whetstone`

More information concerning the Whetstone benchmark is available from `netlib`.

3.4.4 Dhrystones

Written in *Ada*, *C*, and *Pascal* by Reinhold Weicker, the Dhrystone is a synthetic benchmark. Two versions exist: V1.1 and V2.1; results from the latter version are more reliable.

Like the Whetstone benchmark, the use of statements and data types in this benchmark program are based on statistics. However, the figures concerning the use of various programming language features for the Dhrystone benchmark were derived from many published sources in the early 1980's. Further, this benchmark is directed toward determining the performance of system or integer programs, such as compilers; it does not model numerically-intensive applications.

The Dhrystone code consists of many small loops with integer or logic statements. One problem with this benchmark is the smallness of the loops that often fit into the cache of the machine being tested. This means that the results may show a better performance than the user should expect with larger programs. [Weicker 84]

Again, these benchmarks are publically available by ftp from

`netlib@ornl.gov`
`ftp.nosc.mil:pub/aburto`

More information is available from `netlib`.

3.4.5 MIPS

The MIPS rating for a computer can be determined from a set of small programs known as the MIPS benchmark. The ratio of the geometric mean of the performance of these programs to that for the same programs run on the VAX 11/780 provides the MIPS rating. The VAX 11/780 is defined as a MIPS-1 machine, meaning that it is capable of executing one million instructions per second. If a machine is rated at 10 MIPS, it supposedly performs the given benchmarks ten times faster than a VAX 11/780.

These benchmark results can be deceiving because many of the VAX runs were made several years ago. In other words, these ratings compare the execution of programs compiled and run on machines with recent compilers that include all the latest optimizing techniques (see Section 4 for more information of optimizing compilers) against the execution of the same programs compiled with an older compiler on the VAX. If the VAX compiler were updated to include these techniques and the newly compiled programs rerun on the VAX 11/780, it is likely that the VAX execution times would decrease. This, in turn, would reduce the MIPS ratings of the newer machines with good optimizing compilers.

3.4.6 SPECmarks

The System Performance Evaluation Cooperative was created in the 1980's as a subgroup of NCGA (National Computer Graphics Association). The founders of this group included representatives from Hewlett-Packard, DEC, MIPS, and Sun. This group maintains sets of benchmark programs to rate the performance of new machines.

The SPECmark value is the ratio of the geometric mean of the execution time for a selected set of benchmark programs on the given machine to that for the same set of programs on a Vax 11/780, using elapsed time. The first set of benchmark programs is called SPEC89 and contains ten programs. A later set, SPEC92, is made up of twenty programs: six integer programs and fourteen floating-point programs. The SPEC89 result for a machine includes the average speed for both the integer and the floating-point programs; the SPEC92 result is broken into two parts: the SPECint92 result and the SPECfp92 result.

These three benchmarks were designed to test CPU performance. Two later benchmarks were developed by the SPEC group: SDM to test a UNIX Software DevelopMent workload and SFS for benchmarking a Systems level File Server workload.

Unlike other benchmarks, the SPEC benchmark programs are carefully controlled by the personnel within the SPEC group; they run the tests and publish the results in their own periodical. This provides a more uniform and less biased set of results.

The SPECmark values for many machines are published by their manufacturers. Occasionally, output from various SPEC tests shows up on the `comp.benchmarks` newsgroup. Table 2 shows the SPECmark values for some workstations, extracted from the file `/pub/spectable` available from the anonymous ftp ftp site: `ftp.cdf.toronto.edu`. Notice how the difference between the SPECint92 and SPECfp92 values provides more information than is shown by the SPEC89 value.

To see the latest list of SPEC benchmarks, you may retrieve the file, `/pub/spectable`, yourself from the anonymous ftp site: `ftp.cdf.toronto.edu`. To obtain more information on the SPEC benchmark programs, contact

SPEC
c/o NCGA
2722 Merrilee Drive
Suite 200
Fairfax, VA 22031

or send an electronic mail message to `spec-ncga@cup.portal.com`.

3.4.7 Shallow Water Code

The Shallow Water benchmark is a kernel from a *Fortran* weather prediction application program used by the National Center for Atmospheric Research (NCAR). The purpose of the benchmark is to compare the performance of supercomputers on this code. Results are given both in elapsed time and in Mflops [Sadourney 75, Hoffmann et al. 88].

3.4.8 SPICE

SPICE (Simulation Program with Integrated Circuit Emphasis) is a circuit-level simulator developed in the early 1970's at Berkeley. Many versions of it are still being used to assist in the analysis of processor design today. Parts of the code for this simulator are now used for benchmarking purposes to measure a floating-point workload. This is considered to be a real benchmark program, as compared to a kernel benchmark; complete routines are used for the benchmark programs. The results are given as elapsed time or as Mflops.

The most recent version of this benchmark is `spice 3f2` and can be purchased from Berkeley. It is available in both *C* and *Fortran*. To order a software catalog that includes this and other items, send an electronic mail message to `ilpsoftware@berkeley.edu`.

3.4.9 NAS Kernels

A set of benchmarks developed by NASA Ames Research Center for the purpose of comparing the performance of parallel machines is known as the NAS Kernels or the NAS Parallel Benchmarks. This is made up of eight kernels, ranging from an integer sort program to what they call an *embarrassingly* parallel program (since it maps easily to a parallel implementation with almost no serial code). The user is allowed to alter the basic benchmark code to make the best use of the architecture of the given parallel machine.

The results are provided as ratios of the time taken to execute each kernel on the computer being tested to the time taken for the same program executed on a Cray Y-MP with a single processor. Current NAS performance data can be found via the `mosaic` home page:

`http://www.nas.nasa.gov/RNR/Parallel/NPB`

For more information, see [Bailey et al. 94] or contact

NAS Parallel Benchmarks
NAS Systems Division
Mail Stop 2588
NASA Ames Research Center
Moffett Field, CA 94035

4 The Effect of Optimizing Compilers

One of the goals of the original *Fortran* compiler was to optimize the generated assembly code as much as possible. In the late 1950's when *Fortran* was designed, most programmers wrote code in assembly language. To do so, they were required not only to understand the assembly language, but also the organization of the machine on which it ran. Many of these programmers included all types of tricks in their code (modifying instructions in memory, using the same data location for more than one value, storing data within instructions to save memory space, etc.) in order to squeeze out the best performance for both execution speed and memory space. It was difficult for these programmers to believe that a compiler could generate as efficient a program as they could. The early *Fortran* compilers were forced to meet this challenge.

The fact that *Fortran* came close to its goal and has worked to improve its performance ever since is as much of the reason that it is used for high-performance scientific computing as its portability as a high-level language. It quickly became the language for mathematical software, such as LINPACK; currently many commerical and research software libraries exist in *Fortran*. As the language matured, more features were added to make both performance and scientific programming easier and more efficient. This in turn made it even more popular among the scientific computing community.

Code that has been generated by an optimizing compiler should run faster than code that has not been optimized. The better the optimization, the faster the code executes. When any of the benchmark programs listed above are tested on a computer, they must first be compiled. If the compiler generates very good optimizations, the performance of the benchmarks are better than if the compiler generates poor optimizations. As part of recording benchmark results, identification of the compiler and of the level of optimization used for the compilation of the benchmark programs is significant. It is quite possible for two separate benchmarking runs on the same architecture to produce disparate results, simply because different compilers or optimizations are used to create the benchmarking object code.

Recall that the BLAS1 routines may be available in a fast assembly code version on some machines. Other assembly code libraries for common operations exist as well. These might be accessed by the compiler with a library option pointing to the given library of routines; they might also be automatically included when requesting a high level of optimization from a compiler. Compiled programs that include such routines run more efficiently than similar programs with the same routines compiled from high-level language source code.

4.1 Possible Compiler Optimizations

The goal of compiler optimization is to generate efficient assembly code for the high-level language instructions being compiled. This is done at different levels and stages during the compilation process.

One optimization technique is the recognition of *common subexpressions*. Consider the *Fortran* statements taken from an advection code shown in figure 10(a). The variable names and expressions used in this loop closely resemble those in the formulas describing this operation, making the code more understandable for the human reader. The subexpression

```
0.5D0*SIN(2.0D0*PI*(I+J)*H)
```

is used in two of the statements in the loop. A local optimizing stage of the compiler may generate code that first creates a temporary variable containing the common value and then substitutes it into the statements as given in figure 10(b). Simply calculating this common subexpression beforehand and then substituting its computed value into the two statements removes four multiplications, one addition, and one `SIN` function evaluation from the code without altering any of the results of the computation. Since these statements fall within a loop that executes n^2 times, $4n^2$ multiplications, n^2 additions, and n^2 evaluations of the `SIN` function have been eliminated.

Further optimization is possible by examining the loop for more common subexpressions. For instance, the value of `N/2.0D0` is used twice within the loop. So we may add a statement to compute that value and use the new variable name in the original expressions, saving one division per loop iteration. Since the value of `N` does not change during the execution of the loop, it is possible to remove the computation of `N/2.0D0` from the body of the loop and place it ahead of the loop. In this manner, it only executes once, not n^2 times. In fact, it is possible to create statements to compute all of the following subexpressions

```
CSBE1 = 2.0D0*PI*H
CSBE2 = -16*H*H
CSBE3 = N/2.0D0
CSBE4 = E*H
```

and put them before the loop, as the values of these variables are not modified during the loop. This is shown in figure 10(c). By this simple *reordering of*

```
      DO J=0,N-1
         DO I = 0,N-1
            U(I,J)= 0.5D0*SIN(2.0D0*PI*(I+J)*H)
            V(I,J)= -0.5D0*SIN(2.0D0*PI*(I+J)*H)
            PHI(I,J,B1)= EXP(-16*H*H*((I-N/2.0D0)**2 +
     $                (J-N/2.0D0)**2))*SIN(E*I*H)
          ENDDO
      ENDDO
```

(a)

```
      DO J=0,N-1
         DO I = 0,N-1
            CSBEXP = 0.5D0*SIN(2.0D0*PI*(I+J)*H)
            U(I,J)= CSBEXP
            V(I,J)= -CSBEXP
            PHI(I,J,B1)= EXP(-16*H*H*((I-N/2.0D0)**2 +
     $                (J-N/2.0D0)**2))*SIN(E*I*H)
         ENDDO
      ENDDO
```

(b)

```
      CSBE1 = 2.0D0*PI*H
      CSBE2 = -16*H*H
      CSBE3 = N/2.0D0
      CSBE4 = E*H
      DO J=0,N-1
         DO I = 0,N-1
            CSBEXP = 0.5D0*SIN(CSBE1*(I+J))
            U(I,J)= CSBEXP
            V(I,J)= -CSBEXP
            PHI(I,J,B1)= EXP(CSBE2*((I-CSBE3)**2 +
     $                (J-CSBE3)**2))*SIN(CSBE4*I)
         ENDDO
      ENDDO
```

(c)

Figure 10: An example of common subexpressions. (a) The original code in the `initw2d1.f` subroutine used to initialize array values within a larger advection code. (b) Computing one subexpression separately and substituting its value in the other statements. (c) Computing additional subexpressions outside the loop.

statements, the common subexpressions are only computed once, before the loop begins. This eliminates $5(n^2-1)$ multiplications and $2(n^2-1)$ divisions, in addition to the operations already saved by the use of the first common subexpression. Reordering of statements can be used in other ways to reduce execution costs as well.

Loop unrolling is another optimization technique. By repeating the body of a loop several times within that loop, the number of iterations needed for the loop is divided by the number of body repetitions. In turn, this reduces the number of clock cycles spent updating the loop variable and checking for the end of the loop, that is, the *loop overhead*. For example, consider the simple loop shown in figure 11(a). The result of executing this loop should be the same as that result from executing the code displayed in figure 11(b). In the second set of code, the loop from the first example has been unrolled six times. This means that the loop overhead should be one-sixth of that for the first example. The variable, `NN`, is used to handle the case when `N` is not divisible by six; the second loop takes care of the last few elements. Of course, loop unrolling also produces more instructions in the execution code for the loop code segment; this means that there is more instruction space required.

Optimizing compilers keep track of where variables are defined and where they are used in the program. The use of an undefined variable may cause a compiler warning message. Likewise, some variables are defined, but never used. This may be due to an error (as in the case of undefined variables) or may be the result of a program modification. When an *unused variable* is found, an optimizing compiler may remove the entire statement that defines the variable by simply not generating code for it. In other words, unused variables might not be computed.

It is even possible for a compiler to remove an entire loop, if it determines that the result of the computation of that loop is not needed later in the program. For instance, if no elements of the array `A` are used in the program after the loop in the last examples, there is no need to execute that loop. The program executes in a more efficient manner if no code is generated for that loop, that is, if the compiler *removes the loop*. However, if just one element of the array is used later, perhaps just printing out the first element as a double check at the end of the program, the loop is not removed.

A loop may also be replaced by a simpler computation. Consider the loop in the example shown in figure 12(a). Since the number of iterations

```
DO  I=1,N
  A(I) = B(I)
ENDDO
```

(a)

```
NN = (N/6) * 6
DO  I=1,NN,6
  A(I) = B(I)
  A(I+1) = B(I+1)
  A(I+2) = B(I+2)
  A(I+3) = B(I+3)
  A(I+4) = B(I+4)
  A(I+5) = B(I+5)
ENDDO
DO  I=NN+1,N
  A(I) = B(I)
ENDDO
```

(b)

Figure 11: An example of loop unrolling. (a) The original simple loop. (b) The loop unrolled to a depth of six.

```
DO  I=1,100
  X = X + 2
ENDDO
```

(a)

```
X = X + 200
```

(b)

Figure 12: An example of loop replacement. (a) A replaceable loop. (b) A statement that produces output equivalent to that of the loop.

for this loop is constant, the compiler can tell that the value of `X` after the completion of the loop should be `200` more than it was at the beginning of the loop. Thus, it may simply replace the loop by the statement given in figure 12(b). *Replacing the loop* with a single statement eliminates the loop overhead for 100 iterations.

Another trick of optimizing compilers is to replace a procedure call by the procedure code itself; this is called *in-line expansion*. For example, consider the pieces of *Fortran* code given in figure 13(a). Since this subroutine is so simple, it makes sense to put it directly into the program, replacing the CALL statement and removing the subroutine as shown in figure 13(b). This eliminates the cost of a procedure call while producing the same result.

4.2 Assembly Language

The programmer must be aware of all the tricks of optimizing compilers when writing programs to measure performance. Otherwise, the statements to be timed or measured may be combined, moved, modified, replaced, or removed by the compiler. The generated code might not resemble the desired code at all. The best way to double check is to examine the assembly code itself. This is not as difficult as it may seem.

The *Fortran* program displayed in figure 14(a) is created from the *Fortran* statements shown in figures 12(a) and (b). If we compile this with no optimization and look at the assembly code, we should be able to see the difference between the two methods of computing the value for `X`. Figure 12(b) shows the compilation of this program with the `-S` option to produce an assembly code listing file (with a `.s` extension). It also shows the results of an execution of the program to check that the program runs correctly. Both the compilation and run were done on a SUN Sparcstation 10. The assembly code listing contains 118 lines. The code generated for the loop takes several lines and is shown in figure 15(a). The opcode `ld` is a load operation; the first of these loads the value of `200` into floating-point register 1 (`f1`) and the second loads `X` into `f2`. The `fadds` is a floating-point add operation that adds the values of the two floating-point registers and places the result back into `f2`. The contents of `f2` is then stored in the location for `X` in main memory by the `st` operation. The remainder of the code in the loop adds one to the loop index, compares it to the final value for `I`, and branches back to statement `L17` (if the loop is to be repeated) or to statement `L20` (if the

```
...
CALL MYSUB (A, B, N)
...
SUBROUTINE MYSUB (X, Y, M)
INTEGER  I, M
REAL  X(M), Y(M)

DO  I=1,M,2
   X(I) = Y((I+1)/2)
ENDDO

RETURN
END
```

(a)

```
...
DO  I=1,N,2
   A(I) = B((I+1)/2)
ENDDO
...
```

(b)

Figure 13: An example of inline expansion. (a) The subroutine call with the subroutine. (b) Inline expansion replaces the subroutine call.

```
PROGRAM EXAMPL

REAL  X
INTEGER I

X = 0
DO  I=1,100
  X = X + 2
ENDDO
WRITE(*,*)  X

X = 0
X = X + 200
WRITE(*,*)  X

END
```

(a)

```
% f77 -S exampl.f  -o exampl
exampl.f:
 MAIN exampl:
% exampl
    200.000
    200.000
```

(b)

Figure 14: An example *Fortran* program with compilation and execution. (a) The sample program. (b) Compiling and executing the program.

```
L17:
        sethi   %hi(L2000001),%o0
        ld      [%o0+%lo(L2000001)],%f1
        ld      [%l7+-0xffc],%f2
        fadds   %f2,%f1,%f2
        st      %f2,[%l7+-0xffc]
        add     %l6,0x1,%l6
        cmp     %l6,0x64
        bg      L20
        nop
        b       L17
        nop
L20:
```

(a)

```
        sethi   %hi(L2000002),%o0
        ld      [%o0+%lo(L2000002)],%f4
        ld      [%l7+-0xffc],%f5
        fadds   %f5,%f4,%f5
        st      %f5,[%l7+-0xffc]
```

(b)

```
        sethi   %hi(L2000000),%o0
        ld      [%o0+%lo(L2000000)],%f3
        st      %f3,[%l7+-0xffc]
```

(c)

Figure 15: Portions of the SUN Sparcstation 10 assembly code listing for the *Fortran* program given in figure 14. (a) Assembly code generated for the `DO` loop. (b) Assembly code generated to add 200 to `X`. (c) Assembly code to initialize `X` to zero.

loop is finished). The `nop`'s (no operation) are fillers for dead cycles in the operation of the processor.

The code for the addition statement is similar as shown in figure 15(b). Here, the value of `200` is loaded in to `f4`, and `X` is put into `f5`. The result of the addition is also in `f5` and is stored back into `X`. It should be clear that this code takes less cycles.

Furthermore, this example shows that if you have an idea of how the generated code might appear, you can locate the loops and other statements within an assembly code listing. For instance, in both cases, the assembly code statements above are preceded by the statements in figure 15(c). These are the statements that initialize `X` to the value of zero. In both cases, the sample code statements are followed by `call` statements to set up the `WRITE`'s.

This same program is also compiled with an assembly code listing on an SGI Indigo, containing a MIPS processor. This compiler includes the original *Fortran* statements as comments in the assembly code, making it easier to read. The assembly code for the first initialization of `X` and for the `DO` loop is given in figure 16. The operation codes are different, but it is still possible to understand what is going on. The floating-point register operands begin with `$f`, while general purpose register operands only have the `$`. The line numbers from the original *Fortran* program have been included to aid the programmer.

It does take a little practice to read assembly code, but it is not that difficult. You do not need to be able to write assembly code to be able to understand most of it. Manuals explaining the assembly codes for different processors are usually available from the manufacturer, if needed.

4.3 Compiler Optimization Levels

Most compilers have different levels of optimization, from no optimization to as much optimization as possible for the compiler. In some cases, the highest level of optimization may not be perfected; the compiler writers may warn of possibly incorrect results using that level under certain circumstances. The `-O` option ('O' for Optimization) for a UNIX compiler requests the default level of optimization; this level may have been included during the installation of the compiler and may be different on different machines. If the `-O` is immediately followed by an integer, a particular level of optimization is

```
 #   6         X = 0
        li.s    $f4, 0.0000000000000000e+00
        s.s     $f4, 28($sp)
        .loc    2 7
 #   7         DO  I=1,100
        .loc    2 7
        li      $14, 1
        sw      $14, 24($sp)
        .loopno 0 0
$32:
        .loc    2 8
 #   8           X = X + 2
        l.s     $f6, 28($sp)
        li.s    $f8, 2.0000000000000000e+00
        add.s   $f10, $f6, $f8
        s.s     $f10, 28($sp)
        .loc    2 9
 #   9         ENDDO
        lw      $15, 24($sp)
        addu    $24, $15, 1
        sw      $24, 24($sp)
        bne     $24, 101, $32
```

Figure 16: Part of the SGI Indigo assembly code listing for the *Fortran* program given in figure 14. This includes the first initialization of `X` to zero and the `DO` loop. This assembly code listing should be compared to the SPARC assembly code shown in figure 15. At first glance they appear quite different. However, they both use labels (`L17` versus `$32`); it is possible to guess at the meaning of many of the operation codes from the neumonics; and both general purpose registers and floating-point registers are distinguishable as operands.

requested, with `0` or `1` being the lowest optimization level. For instance, the `-O2` option requests the second level of optimization.

When measuring performance of a machine on a certain program or set of programs, the same compiler and the same level of optimization should be used for all compilations. Easy duplication of reported results is important. Therefore, remember to record all the compiler and system parameters associated with any performance measures.

5 Other Architectural Factors

As we already mentioned, the performance of a computer is also affected by architectural features other than the processor in the CPU. In this section, we consider some of those features.

5.1 Memory Hierarchy

The need to access data values and instructions from memory is one bottleneck to computing speed. Large memories tend to be slow to access. The average time to read a data item from a large memory is greater than the time required for arithmetic and logical operations; in fact, it can be a few magnitudes greater. Fast memories tend to be small; they cannot hold all the needed values for a reasonably-sized program. Fast memories are also more expensive than large, slow memories.

To demonstrate the memory bottleneck, consider the following example. Suppose a machine is capable of adding two numbers together within three clock cycles. Also assume that the machine requires five clock cycles to fetch each of the two operands and another five to store the result. This means that an addition on a machine with only a main memory takes more than the three clock cycles; it requires at least 18 cycles.

Computer design seeks to overcome this problem with a hierarchy of memory devices. The idea is to place the data that is accessed more often into small, fast memories while storing the rest in slower memories. The basic memory hierarchy of a computer is broken up into main memory, auxiliary memory, and cache, as shown in figure 17. The arithmetic-logical unit (ALU) of the CPU contains the register file; this may be considered a form of small, fast, expensive memory.

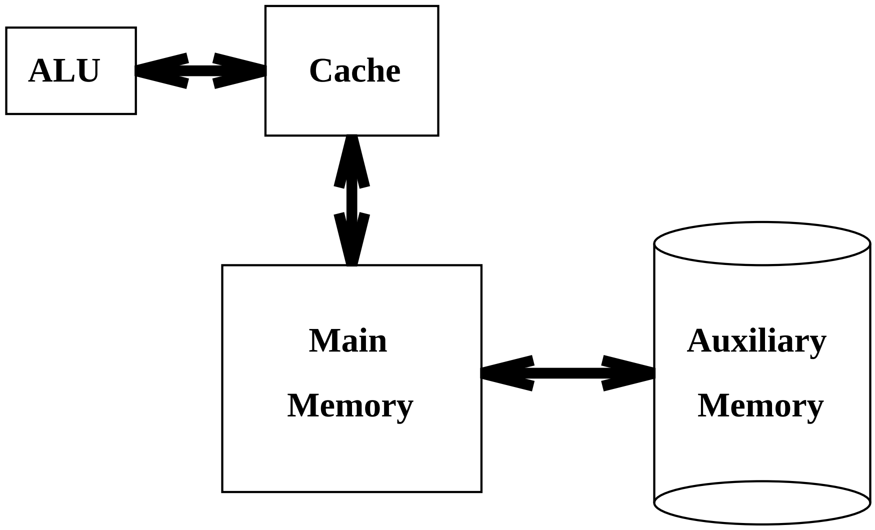


Figure 17: The structure of a basic memory hierarchy.

ALU register accesses are usually completed in one clock cycle. The cost of a cache access is usually 2 to 4 times the cost of accessing a register value. To access data from main memory may require from 5 to 10 CPU clock cycles. And the cost of accessing auxiliary memory may be as much as 20,000,000 nsec. or a few million clock cycles.

When referring to memory hierarchy components, the term *latency* is used to mean response time, and *bandwidth* is used instead of throughput. The basic definition of bandwidth is the maximum number of bits (or bytes) per second that can move along a channel from one computer component to another. This can also be called the communication rate or the transfer rate.

The memory hierarchy is effective because of the *Principle of Locality of Reference*. *Spatial* locality infers that data items stored near each other in the address space of the memory tend to be accessed near each other in time. *Temporal* locality implies that the set of data items referenced over a fixed time window changes slowly with time. This means that the data items being accessed by a program at a given time will probably be accessed again in the near future as might their neighboring data items. Putting the most recently accessed data items into a fast memory so that they can be accessed again quickly should make efficient use of that small memory. Further, the contents of this small, fast memory should be dynamic, slowly changing over the course of the execution of the program.

A *cache* is a small, fast memory that is sometimes placed on the same chip with the CPU. It holds copies of recently accessed words that, according to the principle of temporal locality, are likely to be accessed again. Data items from a cache can be accessed quickly. The reuse of words in the cache saves multiple main memory accesses that require more clock cycles. Values can also be written directly to the cache and copied to main memory at a later time, avoiding the delay often caused by writing to memory.

The size of a block of data that is copied from (or to) the main memory to (or from) the cache in one memory access is called a *cacheline*; a cacheline may contain 8 to 128 bytes of data, depending on the given machine. In other words, a cacheline usually contains more than the one data item being accessed. This means that some of the neighboring data items are also placed into the cache at the same time as the desired data item. According the the principle of spatial locality, these neighboring items may be needed as well.

When a data value is requested from memory, the cache is first checked to see if the value is there. If it is, a *cache hit* has occurred, and the value is obtained from the cache. If the data is not in the cache, this is called a *cache miss*, and the cacheline containing the needed value is fetched from the main memory and put into the cache; this requires the cache table to be updated. Then the requested data value is retrieved from the cache. Any later requests for that data value will be cache hits and can be fetched from the cache. The previous cacheline destroyed by the new cacheline needs to be written out first only if it has been modified and not yet written to main memory.

The *hit ratio* of a process running on a given cache is the average of the number of cache hits divided by the number of accesses:

$$h = \frac{\#hits}{\#accesses}. \tag{2}$$

The *miss ratio* is $1 - h$. These provide a measure for the performance of the cache.

Most computers today have two caches: one for data values and one for instructions. Loops are the backbone of most high-performance scientific computing, and the instruction caches are usually able to store all the instructions for a loop iteration, thus reducing instruction fetch time. Some machines also have two or more levels of caches.

The *main memory* contains data and instructions for each executing program in units of *pages*. The size of a page may be from 256 to 8196 bytes, depending on the machine. The page size is a multiple of the cacheline size for a given machine. The *memory-access time* of a machine is the time that is required for a data value or an instruction to be fetched from the main memory; this is the cost of reading the cacheline containing that data item from main memory into the cache. Writing a cacheline from the cache to main memory usually requires a similar number of cycles.

Most computers today employ *virtual memory*. Executing programs are assigned a limited number of pages in main memory; that memory is being shared by other processes and the operating system. *Auxiliary memory* (usually a disk) is the location of the rest of the data and program instructions for each user, not just those pages currently being used by each program. If a program needs to access some data or instructions not currently in the main memory, we say that a *page fault* has occurred; the program halts while one of its pages is written out to the auxiliary memory and the new page containing the desired data items is read into the main memory. This involves writing out the old page to disk (if it has been modified), reading in the new page from the disk, and updating the page table; all of this takes considerably more time than a simple memory access. Sometimes the system even swaps out the executing program, replacing it with another program that is ready to go, until the new page has been installed.

When the set of data items currently being used by the executing program cannot all fit into the cache at the same time, the process is slowed by the constant fetching and replacing of cachelines containing the needed data. This situation is called *thrashing*. Thrashing may also happen between main memory and auxiliary memory when the number of pages assigned to a process is not great enough to hold the set of pages being used by that process. Hence, the size of a cacheline, the size of the cache, the size of a page, and the number of pages in main memory may all have an effect on the overall performance of the memory hierarchy of the machine. Thus, in addition to the various access times of the different elements of the memory hierarchy, the sizes and units of these components are factors in the performance of the machine itself.

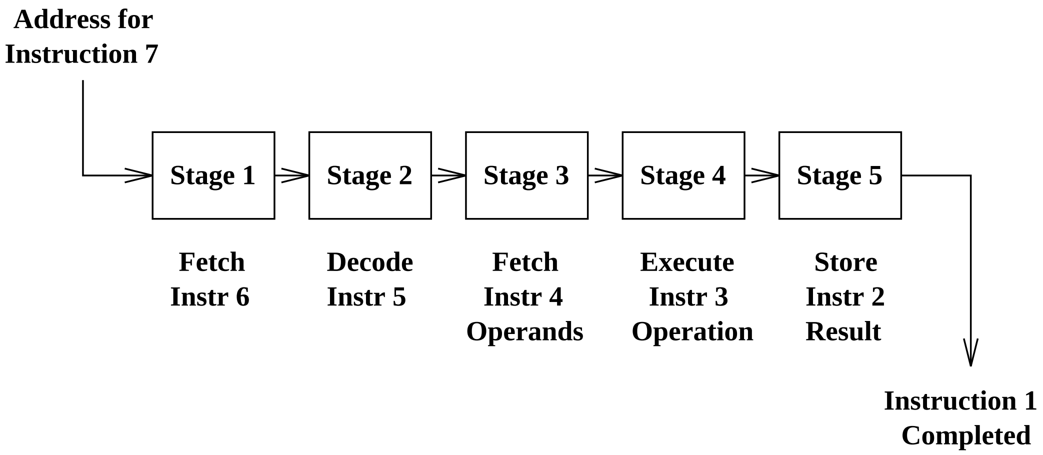


Figure 18: The structure of a basic processor pipeline.

Instr \ Cycle	1	2	3	4	5	6	7	8	9	10	11
1	FI	DI	FO	EX	SR						
2		FI	DI	FO	EX	SR					
3			FI	DI	FO	EX	SR				
4				FI	DI	FO	EX	SR			
5					FI	DI	FO	EX	SR		
6						FI	DI	FO	EX	SR	
7							FI	DI	FO	EX	SR

Figure 19: Overlapping the stages of instruction execution in a basic processor pipeline.

5.2 Pipelined Processors

One of the early methods for speeding up the execution of a program allows the next instruction to be fetched from memory while the current instruction is being executed. Pipelined processors take this idea a bit further. One instruction may be fetched while a second one is being decoded, a third one is fetching its needed operands, a fourth one is performing the desired operation, and a fifth instruction is writing its result. This is diagrammed in figures 18 and 19. In other words, the different parts involved in the execution of one instruction are broken up into stages; each stage may operate concurrently with the other stages on an independent instruction within the processor instruction pipeline. In many respects, this is similar to an assembly line in a factory. If each stage in the pipeline only takes one clock cycle (the smallest unit possible), the processor is capable of executing one instruction per cycle; that is, it has a CPI of one in the ideal case.

Early forms of pipelined processing appeared in the IBM/360 Model 91 and in the CDC 6600. Among the more recent examples of pipelined processors are several RISC (Reduced Instruction Set Computer) processors, including the IBM RS/6000 processor, the line of MIPS processors, the HP-PA processors, and the Sun SPARC processors.

Some CPU architectures contain more than one instruction pipeline and so allow more than one type of instruction to execute at the same time. If a machine is capable of executing more than one instruction per CPU clock cycle, it is called called a *superscalar* computer. For example, the IBM Power2 CPU contains an instruction-pipelined, RISC processor with multiple, separate pipelines for integer operations and for floating-point operations; it is capable of performing two integer operations, two floating-point add operations, and two floating-point multiplications during the same CPU clock cycle.

6 Vector and Parallel Computers

Supercomputers employ various forms of vector and parallel architectures today to achieve their very high speed. The measurement of the performance of these machines needs to include factors related to these types of architectural features. In this section, we discuss their design and their effects on performance.

6.1 Vector Processors

A *vector computer* or *vector processor* contains a set of special arithmetic units known as *vector* or *arithmetic pipelines*.[3] These pipelines are used to efficiently process the elements of vectors or arrays by overlapping the execution of different parts (*stages*) of an arithmetic operation on different elements of the vector. Again, this is similar to an assembly line where the arithmetic operation is being performed in stages; each element in the pipeline is handled, concurrently and independently, at a different stage of the arithmetic operation. The Convex and the Cray Y-MP are vector processors.

Basically, a vector processor deals with vectors instead of scalar values. A *vector* is an one-dimensional array of values, usually held in a *vector register*. For example, vector registers on a Cray Y-MP hold vectors containing 64 data values. The vector registers feed these values into the arithmetic pipelines, allowing the pipeline to perform one arithmetic operation per time unit (usually one clock cycle). When the operation begins, the pipeline must be primed with the first few values in the vector. This accounts for part of the *startup time*, the overhead to get a vector operation flowing through the pipeline.

Scatter and *gather* operations may be used to pick out certain elements in vectors to be processed in the vector pipeline. If these elements are regularly-spaced, the distance between the elements is called the *stride*. For instance, if only the odd-numbered elements are to be used (elements 1, 3, 5, 7, etc.), then the stride of the vector for that operation is two.

The following terms are used to characterize vector processor performance:

- $\mathbf{R_n}$ For a vector processor, the number of Mflops obtainable for a vector of length n.
- $\mathbf{R}_\infty$ The asymptotic number of Mflops for a given machine as the length of the vectors gets large. This means that the startup time would be completely negligible.

[3] More detailed information on this topic can be found in the tutorial on vector processing [Schauble 94].

Characteristic	Cray Model			
	1	**X-MP**	**Y-MP/832**	**Y-MP/C90**
Year Introduced	1976	1983	1989	1990
Max # Procs	1	14	8	16
# Vector Regs	8	8	8	8
Clock Cycle	12.5 nsec	8.5 nsec	6.0 nsec	4.2 nsec
Max Memory	1MB	16 MB	128 MB	16 GB
Word Size	64 bits	64 bits	64 bits	64 bits
Vector Reg Size	64 words	64 words	64 words	64 words

Table 3: A comparison of the characteristics of some Cray vector supercomputers, based on data from [Dongarra 94] and [Hockney & Jesshope 88].

- $\mathbf{n_{1/2}}$ The length, $m = n_{1/2}$, of a vector such that R_m is equal to $R_\infty/2$. This is called the *half-performance length* of the machine and represents the length of a vector needed to produce half the maximum performance for the given machine.
- $\mathbf{n_v}$ The length, n_v, of a vector such that vector operations on that vector are more efficient on the given machine than scalar operations would be.

Table 3 summarizes the characteristics of four Cray models while table 4 compares the performance of those four Crays. These figures are based on data from [Dongarra 94] and [Hockney & Jesshope 88].

For more information on vector processors, see [Schauble 94].

6.2 Parallel Computers

A parallel computer is a machine with two or more connected processors that may operate in parallel. Such a machine is also called a multiprocessor. These machines may be broken up into two main types: MIMD (**M**ultiple-**I**nstruction, **M**ultiple-**D**ata stream) and SIMD (**S**ingle-**I**nstruction, **M**ultiple-**D**ata stream).[4] In other words, the processors of an MIMD multiprocessor

[4]These two terms are from Flynn's taxonomy. The other two classifications are SISD (**S**ingle-**I**nstruction, **S**ingle-**D**ata stream) and MISD (**M**ultiple-**I**nstruction, **S**ingle-**D**ata

Performance	**Cray Model**						
	1	**X-MP**		**Y-MP/832**		**Y-MP/C90**	
# Processors	1	1	4	1	8	1	16
Clock Cycle	12.5 nsec	8.5 nsec		6.0 nsec		4.2 nsec	
Peak Performance Mflops	160	235	940	333	2667	952	15238
LINPACK, $n = 1000$ Mflops	110	218	822	324	2144	902	10780
Shallow Water Mflops	—	—	560	—	1532	—	—
R_∞ $(\vec{x} * \vec{y})$ Mflops	22	70				15000	
$n_{1/2}$ $(\vec{x} * \vec{y})$	18	53				650	

Table 4: A comparison of the performance of some Cray vector supercomputers, partially based on data from [Dongarra 94].

may act independently of one other on different data, while the processors of an SIMD machine perform the same instruction at the same time but on different data. The category of MIMD multiprocessors is also divided into two subtypes: those with distributed memory and those with shared memory.

Benchmarks for the purpose of comparing the performance of various parallel computers are still being developed. The most commonly-used benchmarks include the NAS Kernels and a version of the LINPACK benchmark for solving a linear system of a matrix of size 1000×1000. Both of these benchmarks may be altered by the user to take advantage of architectural features of the given parallel machine. Table 6 provides results from both these benchmarks for some of the parallel machines discussed below. Another set of benchmarks for parallel machines is being developed by the Perfect Club, a group of representatives from manufacturers and universities.

The USENET newsgroup named `comp.parallel` includes discussions of current and new parallel architectures, as well as performance measurements on these machines. Announcements of related conferences and workshops are

stream). The first of these includes the common von Neumann computer; this machine performs one instruction at a time on one set of data. The second set is considered to be empty. [Hockney & Jesshope 88]

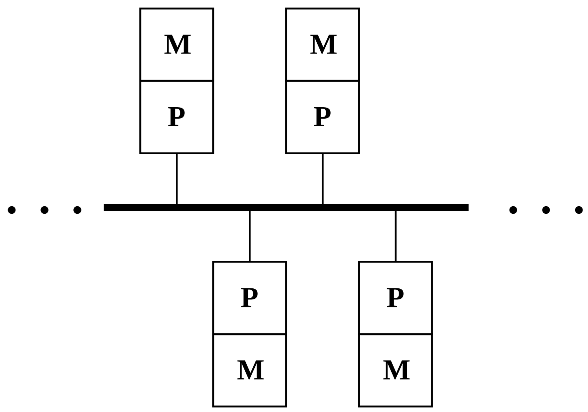


Figure 20: A distributed-memory MIMD multiprocessor.

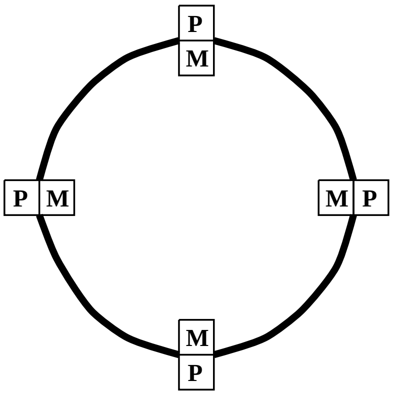
included here as well.

6.2.1 Distributed-Memory MIMD Multiprocessors

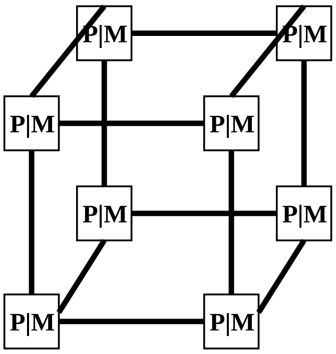
Each of the individual processors of a distributed-memory multiprocessor (DM-MIMD) has a memory unit associated directly with it; that is, each processor has its own local memory. Each of these processors is called a *node* processor and is connected to the other processors by some type of network as displayed in figure 20. To obtain data that is stored in the memory of another processor, a processor must explicitly request that the data be sent to it. The processors of a distributed-memory MIMD multiprocessor send messages to the other processors and receive messages from the other processors in order to distribute and obtain modified data. For this reason, these machines are also called message-passing MIMD computers.

The network joining the processors of a distributed-memory MIMD multiprocessor may be constructed in several ways; some of these are displayed in figure 21. The processors may each be connected to two neighboring processors in the shape of a ring. They may also be connected in a hypercube pattern, as is the Intel iPSC/2. Or they may be connected in a rectangular mesh structure, like the Intel Paragon.

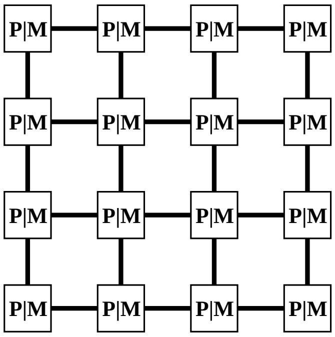
The network of processors that make up a distributed-memory MIMD multiprocessor is often connected to a *front-end* or *host* machine. This is



a. Ring layout.



b. Hypercube layout.



c. Mesh layout.

Figure 21: Different possible layouts for a distributed-memory MIMD multiprocessor.

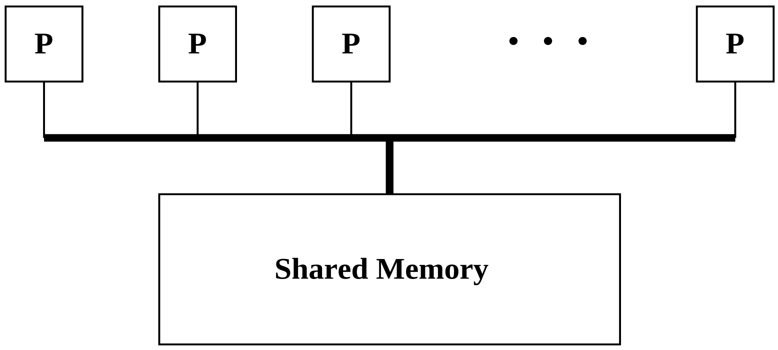


Figure 22: A shared-memory MIMD multiprocessor.

usually a workstation that handles all the input and output for the multiprocessor; it is also used to edit, compile, and store programs. The execution of programs on such multiprocessors is started on the host machine, which passes the program to the individual node processors.

The cost of communicating data between the different processors of a distributed-memory MIMD multiprocessor is an important element affecting the performance of this type of machine. In the early Intel iPSC/1 hypercube, it took about 1000 times as many clock cycles to pass a data value between two processors as it did to access that value from local memory in one processor. While the communication costs have improved since then, so have the costs for computation; thus communication costs are still an important performance factor.

Another example of a distributed-memory MIMD supercomputer is the CM-5 by Thinking Machines Corp. Each of the processors in this multiprocessor contains four vector-processors; hence, this type of machine can also be described as a DM/V-MIMD multiprocessor.

For more information on distributed-memory MIMD multiprocessors, see [Jessup 94]. The theoretical peak performance for some of the machines mentioned above is given in table 5. Table 6 on page 61 compares benchmark results for these machines.

6.2.2 Shared-Memory MIMD Multiprocessors

A shared-memory MIMD multiprocessor (SM-MIMD) has a single memory that may be accessed by any and all of the processors. Early shared-memory MIMD machines provided access to the memory by a bus with a wide bandwidth; this connects all the processors to the memory in a linear manner, as shown in figure 22. Among such machines are the Encore Multimax and the Sequent Balance. The Alliant FX/8 is a similar machine, with the addition of vector processors attached to each processor (SM/V-MIMD). The Convex C3 computers are also shared-memory multi-vector processors, as are the Cray X-MP and the Cray Y-MP with a limited number of processors (4 for the X-MP and 8 for the Y-MP). Again, the theoretical peak performance for some of these machines is shown in table 5, while table 6 compares benchmark results for these machines.

An important problem for shared-memory MIMD multiprocessors is *scalability*: the limitation on the number of processors that can be connected in this manner and still operate efficiently. As the number of processors increases, so does the amount of traffic on the bus connecting the shared-memory and the processors. Increasing the bandwidth of the bus alleviates this to some extent; however, many programs require the use of common variables by most or all the processors. If the value of one variable is updated by each processor, then the processors must line up and wait for serial access to that variable. This is called *contention*. If each of the processors has a private cache, common variables can be kept there, eliminating part of this problem. However, if any of these variables is modified, then each of the caches must be updated to reflect the change; this is called the *cache coherency* problem. Because of bus traffic limitations, most multiprocessors of this design have a maximum of 20 to 30 processors.

6.2.3 Other MIMD Multiprocessors

Some MIMD architectures are built as distributed-memory MIMD multiprocessors, but have software that allows them to behave as shared-memory MIMD multiprocessors (S/DM-MIMD). Data can be passed from the memory of one processor to the memory of another processor without explicit message-passing, thereby allowing simpler programs. The BBN Butterfly was one of the first of this type of machine; the connection of the proces-

Machine	Manufacturer	Type of Multiproc	Max Procs	TPP (Mflops)
iPSC/860	Intel	DM-MIMD	128	7,700
SP1 (9076)	IBM	DM-MIMD	64	8,000
Paragon	Intel	DM-MIMD	4,000	300,000
Cray T3D	Cray Research	DM-MIMD	2,048	307,000
CM-5	Thinking Machines	DM/V-MIMD	16,384	2,000,000
C3800	Convex	SM/V-MIMD	8	960
Symmetry	Sequent	SM-MIMD	30	1,300
Cray Y-MP	Cray Research	SM/V-MIMD	8	2,600
Cray-3	Cray Comp Corp	SM/V-MIMD	16	16,000
KSR1	Kendall Square	S/DM-MIMD	1088	43,500
KSR2	Research	S/DM-MIMD	5000	400,000
MP-1	MasPar	SIMD	16,384	550
MP-2	MasPar	SIMD	16,384	2,400
CM-2	Thinking Machines	SIMD	65,536	31,000

Table 5: A list of distributed-memory MIMD, shared-memory MIMD, and SIMD multiprocessors with their maximum number of processors and their theoretical peak performances with 64-bit floating-point numbers. See [van der Steen 93].

sors was done with a butterfly switch. The KSR1 by Kendall Square is also a member of this group of multiprocessors; this architecture links 32 processors in a ring and connects the rings in a tree structure.

A *cluster* of processors is a group of workstations or PC's connected on a network. These can also be considered distributed-memory MIMD multiprocessors with longer communication times between the processors. The design of the IBM SP1 originated from a cluster of IBM RS/6000 workstations connected in a network. Since all the workstations for this multiprocessor are of the same type, this is called a *homogeneous* cluster. *Heterogeneous* clusters are made up of different types of machines connected together in some fashion. The PVM (Parallel Virtual Machine) language was developed to handle the message-passing details for this type of multiprocessor; it is now available on many distributed-memory MIMD machines, including the Cray T3D. The

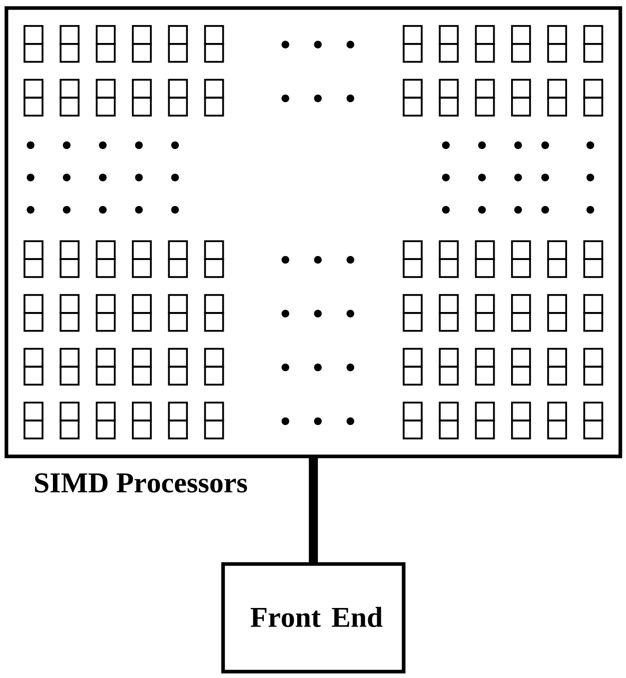


Figure 23: A SIMD multiprocessor.

MPI (Message Passing Interface) language is currently being developed as a standard for the same purpose.

6.2.4 SIMD Multiprocessors

A SIMD multiprocessor directs all its processors to execute the same instruction at the same cycle, but with different data. Like distributed-memory MIMD multiprocessors, SIMD machines are often dependent on a front-end or host machine that dispatches a sequence of instructions to the SIMD machine. The basic structure of this type of machine is similar to that shown in figure 23. Each processor has access to its own private memory containing the data for that processor. Examples of SIMD multiprocessors include the CM-2 from Thinking Machines and the MasPar MP-1 and MP-2. Array processors, such as the DAP, also fall into this category. Table 5 also provides the theoretical peak performance for some SIMD multiprocessors, and table 6 compares benchmark results for these machines. For more information on SIMD multiprocessors, see [Schauble 93].

6.2.5 Performance of Parallel Programs

When considering the performance of a parallel program on a given machine, it is usually compared to the performance of the same program on a single processor of that machine.

Let the symbol T_1 represent the elapsed time of the program run on a single processor. And let T_p represent the time for the program running on p processors of the parallel machine. Then the *speedup* of the program is defined as

$$\textbf{Speedup} = S(p) = \frac{T_1}{T_p}.$$

In an ideal situation, every part of the program could run in parallel, and the speedup would be equal to p. Since this is not often possible, a parallel program is considered to be quite good if its speedup is close to p.

A related parameter is the *efficiency* of a parallel program. This is defined as

$$\textbf{Efficiency} = E(p) = \frac{S(p)}{p}.$$

where p is the number of processors. For a speedup > 1, the values of $E(p)$ should range between $\frac{1}{p}$ and 1; an ideal value for $E(p)$ would be 1, meaning $S(p) = p$ and that the processors are 100% utilized throughout the execution of the program.

A speedup of less than p is expected because not all parts of a program can be run in parallel. For instance, input and output are often best done by a single processor; otherwise, the order of items read or written can be chaotic. Some machines require each processor to pass information or receive information from other processors; this requires communication time, while one processor waits for the other to record and send its values. In a similar manner, not all tasks require the same amount of processing time; some processors may be idle while others are finishing their computations.

Amdahl's Law asserts that speeding up the execution time of one part of a program can at most decrease the entire execution time by the time spent on that part of the program. This means that those operations of a parallel program that must be done sequentially restrict the potential speedup for

Machine	**Clock Cycle** nanosecs	**Number of Procs**	**LINPACK** $n = 1000$ Mflops	**NAS Parallel** **Int Sort**	**Embar Par**
Intel iPSC/860		128	219	0.84	4.91
Intel Paragon	20	128	34(?)	0.86	26.02
IBM SP1	16	64		3.7	20.79
IBM SP2		16		3.68	9.71
CM-5	31.25	32		0.27	5.88
		128		0.96	23.49
		512			90.47
Convex C3800	16.7	4	425		
		8	795		
Convex C4/XA2	7.41	2	1320		
Cray Y-MP/832	6	1	275	1.0	1.0
		8	2144	6.19	7.95
KSR1		32	513	1.1	1.81
		64		1.7	4.9
KSR2	25	32		1.6	5.1
		64		2.94	9.71
MasPar MP-1	83	16,384		1.00	1.82
MasPar MP-2	80	16,384		1.49	5.63
CM-2		65,536		0.77	6.71

Table 6: Selected benchmark results for a group of distributed-memory MIMD, shared-memory MIMD, and SIMD multiprocessors. Performance figures taken from [Bailey et al. 94], [van der Steen 93], and [Dongarra 94].

Amdahl's Law				
	$T_s = 5$ %		$T_s = 50$ %	
p	T_p	$S(p)$	T_p	$S(p)$
1	1.0000	1.0000	1.0000	1.0000
2	0.5250	1.9048	0.7500	1.3333
3	0.3667	2.7273	0.6667	1.5000
4	0.2875	3.4783	0.6250	1.6000
5	0.2400	4.1667	0.6000	1.6667
10	0.1450	6.8966	0.5500	1.8182
20	0.0975	10.2564	0.5250	1.9048
50	0.0690	14.4928	0.5100	1.9608
100	0.0595	16.8067	0.5050	1.9802
200	0.0548	18.2648	0.5025	1.9900
500	0.0519	19.2678	0.5010	1.9960
1000	0.0510	19.6271	0.5005	1.9980

Table 7: Values of T_p and $S(p)$ for $T_s = 5\%$ and $T_s = 50\%$, according to Amdahl's Law.

that program. For example, suppose that 50% of the time to execute a program sequentially can be improved by operations executing in parallel. Even in the best case where the parallel execution is done in virtually zero time, the total execution time is only reduced by one half. Amdahl's law is a formal statement of this observation.

Let T_s represent the time required for the sequential operations of a parallel program. And let T_c represent the amount of time required to complete all the operations that can be done in parallel (or concurrently), but as done sequentially. Thus, the total time for the program to execute on a sequential machine should be computed as

$$T_1 = T_s \ + \ T_c.$$

Now if p is the number of processors that this program can use in parallel, then the total execution time in parallel can be computed as

$$T_p = T_s \ + \ \frac{T_c}{p}.$$

Substituting the values for T_1 and T_p into the previous equation for speedup yields

$$S(p) \ = \ \frac{T_s \ + \ T_c}{T_s \ + \ \frac{T_c}{p}}.$$

For a more detailed example, suppose that T_1 is one unit of time. Let T_s be 5% and let p vary between 1 and 1000. Table 7 displays the values for T_p and $S(p)$ for the different values of p. The two last columns in this table show the same figures for the case when T_s is 50% (the example discussed in the last paragraph). Figure 24 provides plots of these values. Notice that as the number of processors increase, the additional speedup decreases. This is because the speedup has an upper bound of $\frac{1}{T_s}$.

If the speedup of a parallel program exceeds p, the program is called *superlinear*. While Amdahl's Law claims that this should not be possible, some algorithms have achieved this distinction. In particular, several parallel search techniques allow each of the processes to go off in different directions. Some of these processes may be sent off to dead ends, while others are pursuing more profitable paths. As soon as a result is found by one process, it can signal the rest of the processes to terminate their search, thus killing those dead end processes before they would normally have completed on a sequential machine.

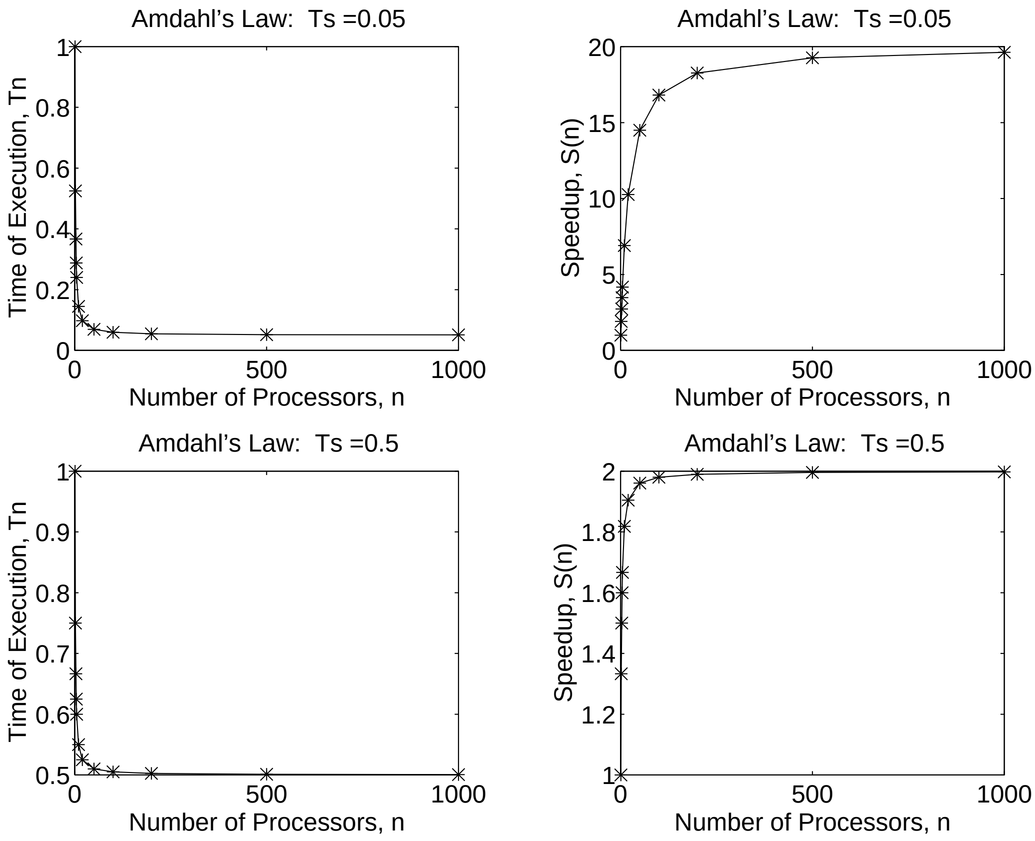


Figure 24: Plots of Amdahl's Law for $T_s = 5\%$ and $T_s = 50\%$ where the number of processors varies from 1 to 1000.

6.2.6 Tools for Parallel Programming

A number of tools exist to help parallel programmers understand and improve the performance of their parallel programs. Most manufacturers of vector and parallel computers provide special operating systems and compilers with extensions that exploit the parallel (and vector) mechanisms of the machine. Both *Fortran 90* [Adams et al. 92] and *HPF* (High Performance Fortran) [Koelbel et al. 94] are designed to consolidate multiple extensions into a uniform version of the *Fortran* programming language, allowing additional operations that could be compiled by different machines into parallel or vector operations.

Debugging tools for parallel programs are currently under research. Tracing and visualization tools assist the programmer in studying the execution of a parallel program. *Schedule* is one such tool. During the execution of a program, it collects information in order to provide data-dependency graphs and visual representations of the flow of given tasks through the processors [Dongarra & Sorensen , Dongarra & Sorensen 87]. These graphs display the allocation of different parallel tasks to different processors and so show where performance might be improved by a reallocation of those tasks.

Another tool called PICL (Portable Instrumented Communication Library) facilitates the collection of traces of send/receive operations in a distributed-memory MIMD machine. These traces can later plotted by a subtool called ParaGraph to show the parallel execution of the traced program. This display can even be animated [Heath et al. 90].

7 Summary

As we have seen, many factors may influence the performance of a computer. Because of this, it is difficult to compare two different architectures. Just determining the speed of the CPU of a given machine may be an intricate task; we must first be able to measure the granularity of the clock used to time the CPU. Common benchmarks and performance units have been developed to assist in the process of comparing machines; yet they must be used with care. Besides the effect of architectural factors such as pipelining and caching, we must also determine whether the compiler used on the benchmarking program has altered the code that performs the measurements. Furthermore, communication and synchronization issues in parallel computers introduce additional factors influencing computer performance.

References

[Adams et al. 92] ADAMS, JEANNE C., WALTER S. BRAINERD, JEANNE T. MARTIN, BRIAN T. SMITH, AND JERROLD L. WAGENER. [1992]. *Fortran 90 Handbook*. Intertext Publications. McGraw-Hill Book Company, New York, NY. ISBN 0-07-000406-4.

[Anderson et al. 92] ANDERSON, E., Z. BAI, C. BISCHOF, J. DEMMEL, J. DONGARRA, J. DUCROZ, A. GREENBAUM, AND S. HAMMARLING. [1992]. *LAPACK User's Guide*. SIAM, Philadelphia.

[Baer 80] BAER, JEAN-LOUP. [1980]. *Computer Systems Architecture*. Computer Society Press.

[Bailey et al. 94] BAILEY, DAVID H., ERIC BARSZCZ, LEONARDO DAGUM, AND HORST D. SIMON. [Mar 1994]. NAS Parallel Benchmark Results 3-94. RNR Technical Report RNR-94-006, NASA Ames Research Center, NASA Ames Research Center.

[DEC 91] Digital Equipment Corporation, Palo Alto, CA 94301. [Dec 1991]. *DECstation 5000 Model 240: Technical Overview*.

[Dongarra & Sorensen] DONGARRA, J. J. AND D. C. SORENSEN. *Schedule Users Guide*. Argonne National Laboratory.

[Dongarra & Sorensen 87] DONGARRA, JACK J. AND DANNY C. SORENSEN. [1987]. SCHEDULE: Tools for developing and analyzing parallel Fortran programs. In JAMIESON, LEAH J., DENNIS B. GANNON, AND ROBERT J. DOUGLASS, editors, *The Characteristics of Parallel Algorithms*, pages 363–394. MIT Press, Cambridge, MA. ISBN 0-262-10036-3.

[Dongarra 94] DONGARRA, J. J. [1994]. Performance of various computers using standard linear equations software. Technical Report CS-89-85, Oak Ridge National Laboratory, Oak Ridge, TN 37831. `netlib` version as of July 1, 1994.

[Dongarra et al. 79] DONGARRA, J. J., C. B. MOLER, J. R. BUNCH, AND G. W. STEWART. [1979]. *LINPACK User's Guide*. SIAM, Philadelphia. ISBN 0-89871-172-X.

[Dowd 93] DOWD, KEVIN. [1993]. *High Performance Computing*. O'Reilly & Associates, Inc., Sebastopol,CA. ISBN 1-56592-032-5.

[Fox et al. 94] FOX, GEOFFREY C., ROY D. WILLIAMS, AND PAUL C. MESSINA. [1994]. *Parallel Computing Works!* Morgan Kaufmann Publishers, Inc., San Francisco, CA.

[Golub & Van Loan 83] GOLUB, G.H. AND C.F. Van LOAN. [1983]. *Matrix Computations*. The Johns Hopkins Press, Baltimore, MD, 1st edition.

[Heath et al. 90] HEATH, M. T., G. A. GEIST, B. W. PEYTON, AND P. H. WORLEY. [Oct 1990]. A user's guide to PICL (ORNL/TM-11616). Technical Report TN 37931, Oak Ridge National Laboratory, Oak Ridge, TN.

[Hennessy & Patterson 90] HENNESSY, JOHN L. AND DAVID A. PATTERSON. [1990]. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA. ISBN 1-55860-069-8.

[Higham 93] HIGHAM, NICHOLAS J. [1993]. *Handbook of Writing for the Mathematical Sciences*. SIAM, Philadelphia, PA. ISBN 0-89871-314-5.

[Hockney & Jesshope 88] HOCKNEY, R. W. AND C. R. JESSHOPE. [1988]. *Parallel Computers 2*. Adam Hilger. ISBN 0-85274-812-4.

[Hoffmann et al. 88] HOFFMANN, G. R., P. N. SWARZTRAUBER, AND R. A. SWEET. [1988]. Aspects of using multiprocessors for meterorological modelling. In HOFFMAN, G.-R. AND D. F. SNELLING, editors, *Multiprocessing in Meteorological Models*, pages 127–196. Springer-Verlag, New York, NY.

[Jessup 94] JESSUP, ELIZABETH R. [1994]. MIMD computing: An introduction. HPSC Course Notes.

[Kane 88] KANE, GERRY. [1988]. *MIPS RISC Architecture*. Prentice-Hall. ISBN 0-13-584749-4.

[Koelbel et al. 94] KOELBEL, CHARLES H., DAVID B. LOVEMAN, ROBERT S. SCHREIBER, JR. GUY L. STEELE, AND MARY E. ZOSEL. [1994]. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. MIT Press, Cambridge, MA. ISBN 0-262-11185-3.

[McMahon 86] MCMAHON, F. H. [Dec 1986]. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA.

[Patterson & Hennessy 94] PATTERSON, DAVID A. AND JOHN L. HENNESSY. [1994]. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, Inc., San Mateo, CA. ISBN 1-55860-281-X.

[Sadourney 75] SADOURNEY, ROBERT. [Apr 1975]. The dynamics of finite-difference models of the shallow-water equations. *Journal of Atmospheric Sciences*, 32:680–689.

[Schauble 93] SCHAUBLE, CAROLYN J. C. [1993]. The Connection Machine (CM-2): An introduction. HPSC Course Notes.

[Schauble 94] SCHAUBLE, CAROLYN J. C. [1994]. Vector computing: An introduction. HPSC Course Notes.

[Smith et al. 76] SMITH, B.T., J.M. BOYLE, J.J. DONGARRA, B.S. GARBOW, Y. IKEBE, V.C. KLEMA, AND C.B. MOLER. [1976]. *Matrix Eigensystem Routines — EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science*. Springer-Verlag, 2nd edition.

[Stone 87] STONE, HAROLD S., editor. [1987]. *High-Performance Computer Architecture*. Addison-Wesley Publishing Company, Reading, MA, 2nd edition. ISBN 0-201-52688-3.

[van der Steen 93] VAN DER STEEN, AAD J. [Dec 1993]. An overview of (almost) available parallel systems. Technical report, Stichting Nationale Computer Faciliteiten, The Netherlands. Third revised edition, `netlib`.

[Weicker 84] WEICKER, REINHOLD P. [Oct 1984]. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030.