$(r_\infty, n_{\frac{1}{2}}, s_{\frac{1}{2}})$ MEASUREMENTS ON THE 2-CPU CRAY X-MP

Roger W. Hockney
Computer Science Department, Reading University,
Whiteknights, Reading, Berks, U.K.

Summary: We report performance measurements made on the 2-CPU Cray X-MP at ECMWF, Reading. Vector (SIMD) performance on one CPU is interpreted by the two parameters $(r_\infty, n_{\frac{1}{2}})$, and we find for dyadic operations using FORTRAN r_∞ = 70 Mflop/s, $n_{\frac{1}{2}}$ = 53 flop. All vector triadic operations produce r_∞ = 107 Mflop/s, $n_{\frac{1}{2}}$ = 45 flop; and a triadic operation with two vectors and one scalar gives r_∞ = 148 Mflop/s and $n_{\frac{1}{2}}$ = 60 flop. MIMD performance using both CPUs on one job is interpreted with the two parameters $(r_\infty, s_{\frac{1}{2}})$, where $s_{\frac{1}{2}}$ is the amount of arithmetic that could have been done during the time taken to synchronize the two CPUs. We find, for dyadic operations using the TSKSTART and TSKWAIT synchronization primitives, that r_∞ = 130 Mflop/s and $s_{\frac{1}{2}}$ = 5700 flop. This means that a job must contain more than ~6000 floating-point operations if it is to run at more than 50% of the maximum performance when split between both CPUs by this method. Less expensive synchronization methods using LOCKS and EVENTS reduce $s_{\frac{1}{2}}$ to 4000 flop and 2000 flop respectively. A simplified form of LOCK synchronization written in CAL code further reduces $s_{\frac{1}{2}}$ to 220 flop. This is probably the minimum possible value for synchronization overhead on the Cray X-MP.

1. INTRODUCTION

It has been recognised for years (Calahan, 1977; Hockney, 1977; Calahan and Ames, 1979) that the single parameter Mflop/s (megaflops) is inadequate to measure the performance of a vector computer, because it takes no account of the vector startup overhead. Nevertheless there has been a reluctance to use a second parameter to overcome this deficiency, and manufacturers

still do not publish a two-parameter description of the performance of their vector pipelines, even though some (notably Cray, Inc.) could do so to their competitive advantage. The two-parameter (r_∞, $n_{\frac{1}{2}}$) description introduced by Hockney and Jesshope (1981) and Hockney (1983) is based on measuring the importance of the startup overhead in terms of how much useful arithmetic (in fact $n_{\frac{1}{2}}$ floating-point operations) could have been done during the time of the overhead. In a similar spirit this description has been extended (Hockney, 1985) to MIMD computing by measuring the overhead of synchronizing multiple instruction streams in terms of $s_{\frac{1}{2}}$: the amount of useful arithmetic that could have been during the time taken for synchronization. The three parameters r_∞, $n_{\frac{1}{2}}$ and $s_{\frac{1}{2}}$ are defined mathematically in Section 2, and measurements of their values on the 2-CPU Cray X-MP at ECMWF are reported in Section 3.

2. THE PARAMETERS (r_∞, $n_{\frac{1}{2}}$, $s_{\frac{1}{2}}$)

An SIMD computer is one in which a single instruction initiates many identical operations on multiple data items, in short one in which vector instructions are included. The most successful SIMD computers execute a vector instruction by pipelining the successive elements of the vectors (usually one per clock period) through a high-performance multi-stage pipeline. The Cray X-MP is an example of such a computer with a six-stage floating-point adder and a seven-stage floating-point multiplier. The performance of a vector pipeline can be characterized by measuring the time to execute a single vector instruction, t, as a function of the length of the vector, n, and fitting the results to the linear relationship

$$t = r_\infty^{-1}(n + n_{\frac{1}{2}}) \tag{1}$$

where:

r_∞ - is the asymptotic (i.e. maximum) performance in millions of floating-point operations per second (Mflop/s),

$n_{\frac{1}{2}}$ - the half-performance length, is the vector length necessary to achieve half the asymptotic performance.

Expressing Eqn. (1) as a startup time in microseconds, t_0, and a time per result, τ, we have

$$t = t_0 + n\tau \qquad (2)$$

where $t_0 = n_{\frac{1}{2}}/r_\infty$ and $\tau = r_\infty^{-1}$.

Comparing Eqns. (1) and (2) one sees that $n_{\frac{1}{2}}$ is the number of floating-point operations which could have been done in the time of a vector startup. The parameter $n_{\frac{1}{2}}$ therefore measures the importance, in terms of lost floating-point operations, of vector startup to the user. Not surprisingly one finds that, within the approximations of this timing model, $n_{\frac{1}{2}}$ is the parameter that determines the best choice of vector algorithm on a particular computer (see Hockney, 1983, 1984). When the vector length equals $n_{\frac{1}{2}}$ then the two terms in Eqn. (1) are equal, and half the time is being used to perform useful arithmetic (first term) and half the time is being lost in vector startups (second term). In this case the average performance is obviously only 50% of the maximum.

Using Eqn. (1) we find the average performance, r, as a function of vector length to be

$$r = n/t = r_\infty/(1 + n_{\frac{1}{2}}/n) \qquad (3)$$

which gives the functional form of the approach of the average Mflop/s to the maximum Mflop/s, r_∞, quoted by the manufacturer, as the vector length increases. One should note that Eqn. (3) demonstrates a slow approach to the asymptotic performance. By definition a vector length of $n_{\frac{1}{2}}$ is

required to produce an average performance of half the maximum, but also a vector length of ten times $n_{\frac{1}{2}}$ is required to reach 91% of the maximum performance. Equation (3) also shows how the long tables of average performance against vector length often published by computer manufacturers, can be represented much more compactly by quoting equivalently the values of the two-parameters $(r_\infty, n_{\frac{1}{2}})$.

The values of r_∞ and $n_{\frac{1}{2}}$ are likely to depend to some extent on the software being used (e.g. assembler code or FORTRAN) and the types of vector operation (e.g. dyadic, triadic, register-to-register or memory-to-memory). For an accurate description separate pairs of values need to be measured for these different cases. The timing relation (1) can also be used to estimate the time, T_v, for the vector part of an algorithm; that is to say the part that is executed using vector instructions. If the parameters r_∞ and $n_{\frac{1}{2}}$ are approximately constant for the code in question then

$$T_v = r_\infty^{-1}(s_v + n_{\frac{1}{2}}q) \qquad (4)$$

where:

s_v - is the total amount of useful arithmetic (i.e. floating-point operations between pairs of numbers) in the vector part of the algorithm

q - is the number of vector instructions comprising the vector part of the algorithm.

If the parameters r_∞ and $n_{\frac{1}{2}}$ vary too much to be considered constant over the whole vector part of the algorithm, then the vector operations must be grouped so that the parameters are constant within each group. Equation (4) may then be applied to each group using a different pair of values $(r_\infty, n_{\frac{1}{2}})$ for each group.

The performance of scalar instructions, which initiate only a single floating-point operation, can similarly be characterized by the two-parameters (r_∞, $n_{\frac{1}{2}}$), and one finds that usually the value of $n_{\frac{1}{2}}$ is sufficiently small that it can be ignored. Hence the time for the scalar part of an algorithm is given by

$$T_s = r_{\infty s}^{-1} s_s \qquad (5)$$

where:

s_s - is the number of floating-point operations between pairs of numbers in the scalar part of the algorithm

$r_{\infty s}$ - is the performance in Mflop/s of scalar instructions

Combining Eqns. (4) and (5), we have the time, T, for the complete algorithm

$$T = T_v + T_s \qquad (6)$$

Usually one finds that the scalar performance is less than the vector performance by a factor of about ten, showing the importance of executing as much of the arithmetic as possible in vector instructions - that is to say achieving a high level of vectorization.

The critical path through a multi-instruction-stream (i.e. MIMD) program can be considered as a sequence of work segments, between which program synchronization must occur. That is to say, all work must be completed on a segment before the next can begin. Within a segment, however, several independent tasks exist that may be assigned to different instruction streams (i.e. executed by different CPUs in the Cray X-MP). For synchronization to take place correctly it is necessary for the control program to initiate each instruction stream, and to recognize when all the instruction streams have finished. The time taken to perform these

operations is called the synchronization overhead. It is important in multi-instruction stream programming to be aware of this overhead, and to be able to assess its magnitude quantitatively. The parameter $s_{\frac{1}{2}}$ has been introduced for this purpose (Hockney and Snelling 1984, Hockney 1985), and measures the synchronization overhead in terms of how many floating-point operations could have been in the time of the overhead. Accordingly the time, t, to execute a work segment on multiple CPUs becomes

$$t = r_\infty^{-1}(s_i + s_{\frac{1}{2}}) \qquad (7)$$

where:

s_i - is the total number of floating-point operations between pairs of numbers in the ith work segment

If the critical path through an MIMD algorithm comprises a sequence of q work segments, each obeying Eqn. (7), then the total time, T, for an MIMD algorithm becomes

$$T = r_\infty^{-1}(s + s_{\frac{1}{2}}q) \ , \quad s = \sum_{i=1}^{q} s_i \qquad (8)$$

In Eqn. (8) we have implicitly assumed that it is possible to schedule the work, s, between the multiple instruction streams in such a way that each stream has the same amount of work to do. This would be perfect scheduling, or load balancing. If this is not possible then the effect of imperfect scheduling can be included into Eqn. (8) by introducing the average efficiency, $\bar{E}_p$, as defined by Kuck (1978).

$$T = r_\infty^{-1}(s/\bar{E}_p + s_{\frac{1}{2}}q) \qquad (9)$$

As with the case of the vector parameter $n_{\frac{1}{2}}$, Eqn. (6) shows that when $s = s_{\frac{1}{2}}$, half the time is spent on useful arithmetic and half on synchronization overhead, resulting in an average performance of half the maximum, r_∞. Thus $s_{\frac{1}{2}}$ is a yardstick that can be used to judge whether a

work segment is large enough to be worth dividing between multiple CPUs. As with the vector case, the work in a segment must exceed ten times $s_{\frac{1}{2}}$ in order to achieve more than 91% of the maximum performance. In the next section we describe some simple benchmarks that have been used to measure values for the three parameters r_∞, $n_{\frac{1}{2}}$ and $s_{\frac{1}{2}}$ on the ECMWF 2-CPU Cray X-MP.

3. MEASUREMENTS

3.1 Values of (r_∞, $n_{\frac{1}{2}}$) for one CPU

Figure 1 gives the program used to measure r_∞ and $n_{\frac{1}{2}}$ on one CPU of the Cray X-MP. Initially the overhead of making the timing measurement is obtained by measuring the time TØ for two successive calls to the timing routine SECOND. This overhead is subsequently subtracted from all measurements. The vector length, N, is varied in steps of two, up to a maximum of 400, and for each vector length NREPEAT = 100 trials are made. The minimum time, TMIN, maximum time, TMAX, and average time (TSUM/NREPEAT) of the 100 trials are recorded. The DO-1Ø loop is replaced by a vector instruction when compiled, and is the subject of the timing, statement 10 being changed for the different types of loops considered below. Values of (r_∞, $n_{\frac{1}{2}}$) are obtained by fitting the best straight line through the values of TMIN versus N as is shown in Fig. 2. The asymptotic performance, r_∞, is the inverse slope of this line, and the half-performance length, $n_{\frac{1}{2}}$, is the negative intercept of the line with the vector-length axis. In the case of triadic operations TMIN is divided by two, so that the time axis correctly records the time <u>per</u> vector operation. In a multi-user operating system it is necessary to make multiple trials at each vector length and take the least time, in order to minimize interference from other jobs. One hundred trials appears sufficient for this purpose, as the resulting values in Fig. 2, except for a few rogue points, are quite well clustered along the timing line. The results obtained are

```
      PROGRAM MULT
      PARAMETER (NMAX = 4ØØ)
      PARAMETER (NMAXA = NMAX, NMAXB = NMAX, NMAXC = NMAX)
      PARAMETER (NREPEAT = 1ØØ)
      DIMENSION B(NMAXB), C(NMAXC), A(NMAXA)
      DATA B/NMAX*1.Ø/, C/NMAX*1.Ø/
      DATA S/1.Ø/

      CALL SECOND(T1)
      CALL SECOND(T2)
      TØ = T2-T1

      DO 2Ø N = 2, NMAX,2
      TMIN = 1ØØØØØØ
      TMAX = -1ØØØØØØ
      TSUM = Ø

      DO 111 JR = 1, NREPEAT
      CALL SECOND(T1)
CDIR$ IVDEP
      DO 1Ø I = 1, N
 1Ø      A(I) = B(I)*C(I)
      CALL SECOND(T2)

      T = T2-T1-TØ
      TMIN = MIN(TMIN,T)
      TMAX = MAX(TMAX,T)
      TSUM = TSUM + T
111   CONTINUE

      WRITE (6,1ØØ) N, TMIN, TMAX, TSUM/NREPEAT
 2Ø   CONTINUE
1ØØ   FORMAT ('  N: ', I4, 3F15.8)
      STOP
      END
```

Fig. 1 The program used to measure (r_∞, $n_{\frac{1}{2}}$) on a single CPU of the Cray X-MP. The statement 1Ø is changed for the different cases.

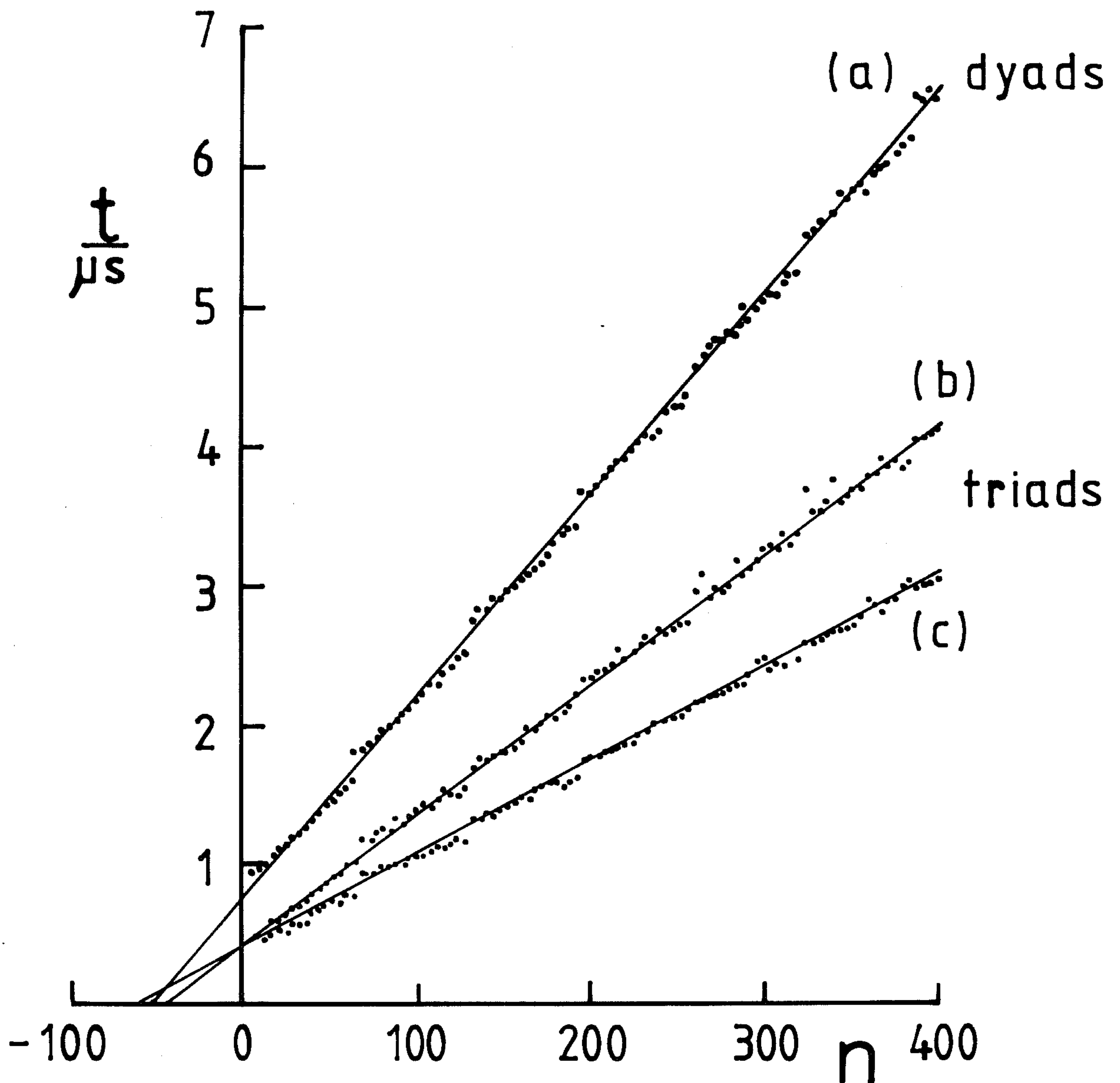

Fig. 2 Measurement of r_∞ and $n_{\frac{1}{2}}$ on one CPU of the Cray X-MP. Time, t, as a function of vector length, n, for a single vector operation. The asymptotic performance, r_∞, is the inverse slope of the best fit line, and the half-performance length, $n_{\frac{1}{2}}$, is the negative intercept of the line with the n-axis. (a) dyadic operations $\underset{\sim}{A} = \underset{\sim}{B}*\underset{\sim}{C}$, (b) all vector triadic operation $\underset{\sim}{A} = \underset{\sim}{D}*\underset{\sim}{B} + \underset{\sim}{C}$, (c) Cyber 205 triad $\underset{\sim}{A} = s*\underset{\sim}{B} + \underset{\sim}{C}$.

given in Table I.

Table I - Measured values of r_∞ and $n_{\frac{1}{2}}$ on a single CPU of a Cray X-MP for memory-to-memory operations. Cray-1 values are in parentheses.

Operation: Statement 10	r_∞ Mflop/s	$n_{\frac{1}{2}}$ flop	t_0 μs
Dyadic A(I) = B(I)*C(I) (Cray-1 values)	70 (22)	53 (18)	0.75 (0.82)
All vector Triad A(I) = D(I)*B(I) + C(I)	107	45	0.42
Cyber 205 Triad A(I) = s*B(I) + C(I)	148	60	0.40
Scalar code A(I) = B(I)*C(I)	5	4	0.80

The first three cases in Table I are measurements of vector instructions. The dyadic case uses only a single vector pipeline with all vectors stored in main memory, and is to be compared with values of $r_\infty = 22$ Mflop/s and $n_{\frac{1}{2}} = 18$ flop previously obtained on the Cray-1 (Hockney and Jesshope, 1981). We find a three fold increase in r_∞ due, primarily to the provision of three memory ports on the Cray X-MP compared with one on the Cray-1. The startup time in microseconds, $t_0 = n_{\frac{1}{2}}/r_\infty$, has not changed significantly between the two machines: the extra complexity of memory access on the X-MP being compensated by a reduction in clock period. However the importance of this overhead, which is what $n_{\frac{1}{2}}$ measures, is three times greater on the X-MP, because three times as much arithmetic could have been done during this time, as compared to the Cray-1. The maximum rate at which one floating-point pipeline can deliver results is one result per clock period of 9.5 ns, that is to say 105 Mflop/s. The

measured value of r_∞ = 70 Mflop/s is less than this because of the time taken to refill the vector registers from main memory. Because the vector registers hold 64 elements, this is an overhead that is incurred every 64 elements, and is just visible in Fig. 2.

The second two cases are measurements for triadic vector operations, which involve the chaining of two vector instructions, and the simultaneous use of both the floating-point multiply and add pipelines. At best we can expect a doubling of r_∞ which is achieved if one of the arguments is a scalar, but is not achieved in the all vector case. The value of $n_{\frac{1}{2}}$ is not materially altered in the triad cases, however the startup time in microseconds is halved because a single startup of 0.8μs is shared between two instructions.

For comparison purposes, we have run the dyad benchmark with instructions to the compiler to use only scalar instructions, and obtain r_∞ = 5 Mflop/s and $n_{\frac{1}{2}}$ = 4 flop. The startup time t_0 remains at 0.8μs but this is now of negligible importance because the arithmetic performance is about twenty times slower than when a vector instruction is used. This is made clear by the fact that $n_{\frac{1}{2}}$ is reduced to a negligible value compared to any vector lengths that are likely to be used, thus substantiating the statement in Section 2 that $n_{\frac{1}{2}} \cong 0$ for scalar code.

3.2 Values of (r_∞, $s_{\frac{1}{2}}$) for two CPUs

Four methods of synchronizing the operation of the two CPUs of a Cray X-MP on a single job have been considered, and the programs used are given in Figs. 3 to 6. They are the use of the TSKSTART and TSKWAIT primitives which we refer to as the TASKS method; the use of the LOCKON and LOCKOFF primitives which we refer to as the LOCKS method; the use of the EVPOST and EVWAIT primitives which we refer to as the EVENTS method; and finally

```
      PROGRAM MULTI
      COMMON/GLOBAL/A(4ØØ), B(4ØØ), C(4ØØ)
      DIMENSION IDT(2)
      EXTERNAL DOALL
      DATA B/4ØØ*1.Ø/, C/4ØØ*1.Ø/

      NMAX = 4ØØ
      IDT(1) = 2

      T1 = 9.5E-9*RTC(DUM)
      T2 = 9.5E-9*RTC(DUM)
      TØ = T2-T1

      DO 2Ø N = 2, NMAX, 2
      T1 = 9.5E-9*RTC(DUM)
      NHALF = N/2,
      NH1 = NHALF + 1

      CALL TSKSTART (IDT,DOALL, NH1, N)
      CALL DOALL (1, NHALF)
      CALL TSKWAIT (IDT)

      T2 = 9.5E-9*RTC(DUM)

      T = T2-T1-TØ
      WRITE (6, 1ØØ) N, T
 2Ø   CONTINUE

1ØØ   FORMAT (' N: ', I4, 4X, 'TIME IN SECONDS:' F16.12)
      STOP
      END

      SUBROUTINE DOALL (N1, N2)
      COMMON/GLOBAL/A(4ØØ), B(4ØØ), C(4ØØ)

      DO 1Ø I = N1, N2
 1Ø      A(I) = B(I)*C(I)

      RETURN
      END
```

Fig. 3 Program for measuring r_∞ and $s_{\frac{1}{2}}$ when a job is split between the two CPUs of the Cray X-MP, using the TASKS method of synchronization.

the use of a simplified LOCKS method written in CAL code. In all cases the programs were run on the computer in stand-alone mode, and timing was performed using the real-time clock function RTC(DUM). In this way we ensure that the second physical CPU is assigned to the second logical CPU in the programs, and that we are measuring the wall-clock time for the complete job.

In the TASKS method (Fig. 3), after calling the timer (RTC) at the start of the measurement (TI), the second CPU is given a copy of the subroutine DOALL by the TSKSTART statement, and begins to execute it. The first CPU, which is performing the control program MULTI, then executes another copy of the subroutine DOALL in the CALL DOALL statement. The TSKWAIT statement ensures that both CPUs have finished their share of the work before the timer is called again to record the end of the measurement (T2). The parameters to DOALL are used to ensure that the two CPUs do different elemental operations (N/2 each) from the total of N operations. In this method the overhead of starting a new task occurs at every FORK into a work segment that is divided between the two CPUs, and is therefore included in the measured time. By choosing an element-by-element vector multiply for even vector lengths, as the work to be done in a work segment, we ensure that the workload of N/2 floating-point operations can be equally balanced between the two CPUs, so that the efficiency of scheduling $E_p = 1$. Hence the measured time, t, can be compared with the standard form of Eqn. (7)

$$t = r_\infty^{-1}(s + s_{\frac{1}{2}}) \qquad (10a)$$

or $$t = t_0 + s/r_\infty \qquad (10b)$$

where $s = N$ of the program in Fig. 3, is the total amount of work split between the two CPUs,

and $t_0 = s_{\frac{1}{2}}/r_\infty$ is the synchronization overhead time in seconds

Table II - Measured values of r_∞ and $s_{\frac{1}{2}}$ when dyadic memory-to-memory operations are split between two CPUs on the Cray X-MP22. The overhead is separately measured using TASKS, LOCKS, EVENTS and CAL code for synchronization. $t_0 = s_{\frac{1}{2}}/r_\infty$ is the synchronization overhead in microseconds. Equivalent values for a single PEM Denelcor HEP1 computer are given in parentheses (Hockney and Snelling, 1984; Hockney 1985).

METHOD	r_∞ Mflop/s	$s_{\frac{1}{2}}$ flop	t_0 μs	$\pi_0 = t_0^{-1}$ k/s
TASKS (HEP1)	130 (1.7)	5700 (820)	45 (490)	22 (2)
LOCKS	140	4000	28	36
EVENTS (HEP1)	140 (1.7)	2000 (230)	14 (140)	71 (7)
simplified LOCKS CAL code	110	220	2	500

Note: results are deliberately rounded to two significant figures only. Greater precision would suggest spurious accuracy.

The measured time fits the formula

$$t = 45 + 3.2\ s/400\ \mu s \qquad (11)$$

which leads to the values of r_∞ and $s_{\frac{1}{2}}$ given in Table II.

We also show in Table II the result of exactly the same measurements performed on a single PEM, Denelcor HEP1 (Hockney and Snelling 1984, Hockney 1985). The comparison raises some important issues concerning overheads and their interpretation. One sees that, although the overhead time in seconds, t_0, on the Cray X-MP is only about one tenth that of the Denelcor HEP1, the value of $s_{\frac{1}{2}}$ of the Cray X-MP is almost ten times more than that of the HEP1. So which is the "best" machine from the overhead point of view? If we interpret "best" to mean least execution time, then no single parameter - neither t_0 alone nor $s_{\frac{1}{2}}$ alone - can answer this question. It is necessary to know two parameters, either the pair $(r_\infty, s_{\frac{1}{2}})$ or the pair (r_∞, t_0). Furthermore Eqns. (10) show that the answer may depend on the amount of work, s, that is split between the CPUs, which is often referred to as the grain of the MIMD calculation. In analogy with Eqn. (3) for vector calculations, one has from Eqn. (10a)

$$r = s/t = r_\infty/(1 + s_{\frac{1}{2}}/s) \qquad (12)$$

for the average performance, r, as a function of grain size, s. Taking the limit of large grain size ($s/s_{\frac{1}{2}} >> 1$), we have $r \rightarrow r_\infty$. Thus r_∞ is the parameter that characterizes the performance of large-grain MIMD programs. However, in the limit of small grain-size ($s/s_{\frac{1}{2}} << 1$), we have the performance

$$r = r_\infty s/s_{\frac{1}{2}} = \pi_0 s \qquad (13)$$

where $\pi_0 = t_0^{-1} = r_\infty/s_{\frac{1}{2}}$ is called the specific performance. Thus π_0, rather than r_∞, characterizes the performance of small-grain MIMD programs. Using the numbers in Table II, one finds that both the small and large-grain performance of the Cray X-MP is greater than that of the HEP1.

In this case, what then is the point of the parameter $s_{\frac{1}{2}}$, which suggests that in some sense the HEP1 is better, not worse, than the Cray X-MP at synchronization? The point is that, for any given grain size, the HEP1 makes more efficient use of the available computing capability: that is to say, given s, the fraction of the maximum Mflop/s that is actually achieved is always higher on the HEP1 than on the Cray X-MP. This follows from Eqn. (12), because $s_{\frac{1}{2}}$ is smaller on the HEP1, and therefore $s/s_{\frac{1}{2}}$ is always larger. Put another way, if the HEP1 were implemented in the same technology as the Cray X-MP, that is to say if it had the same clock period, its r_{∞} would become equal to that of the Cray X-MP, but its value of $s_{\frac{1}{2}}$ would remain unchanged at about one tenth of that of the Cray X-MP. Thus one may say that $s_{\frac{1}{2}}$ is a characterization of the computer architecture (in the general sense of including the efficiency of the synchronization software), whereas r_{∞} is a characteristic of the technology in which the architecture is implemented. In this sense the $(r_{\infty}, s_{\frac{1}{2}})$ characterization of a computer nicely separates architecture from technology. This separation is also evident from the fact $s_{\frac{1}{2}}$ does not contain any physical units, whereas r_{∞} contains inverse seconds in its units.

The conclusion from the comparison is that the HEP1 is architecturally better at synchronization than the Cray X-MP, however the latter is better in absolute terms because of its faster circuitry. The Denelcor HEP2 is said to be architecturally similar to the HEP1, but implemented in high-speed technology. This machine could therefore prove also to be better than the Cray X-MP at synchronization in absolute terms.

In other cases of comparison one may find that one computer has the better small-grain performance and the other the better large-grain performance. It is worth noting that since the vector length equation (1) for $n_{\frac{1}{2}}$ is

identical in form to the grain size equation (7) for $s_{\frac{1}{2}}$, vector length n in SIMD programs and grain size s in MIMD programs are analogous quantities. Hence the previous statement about MIMD performance is analogous to saying of two SIMD computers that one has the better short-vector performance whilst the other has the better long-vector performance.

The second virtue of the $s_{\frac{1}{2}}$ parameter for measuring the synchronization overhead, is that it is the yardstick by which a programmer may judge the grain at which to multi-task a program. If we regard 50% of the asymptotic performance rate, r_{∞}, as the minimum acceptable efficiency, then we know that the grain size s must exceed the $s_{\frac{1}{2}}$ of the computer before multi-tasking is worthwhile. Thus the figures in Table II tell us that the HEP1 is suitable for multi-tasking problems with about ten times finer grain than is the Cray X-MP. Furthermore, $s_{\frac{1}{2}}$ is measured in units of floating-point operations, to which a numerical analyst devising an algorithm can directly relate, because he knows the number of such operations in different parts of his algorithm. If however the overhead is measured in physical units, e.g. by t_0 in microseconds, then these are not the units in which the complexity of an algorithm is most naturally expressed or known.

The next least expensive method of synchronization proves to be the LOCKS method, the code for which is shown in Fig. 4. In this case we observe (Table II) an $s_{\frac{1}{2}} = 4000$ flop about 2/3 of the value found for the TASKS method. The code for synchronization using the EVENTS method is given in Fig. 5, and we find this to be half as expensive as the LOCKS method with $s_{\frac{1}{2}} = 2000$ flop. In order to determine the least overhead possible, a simplified form of the LOCKS method has been programmed in CAL by John Larson of Cray Research Inc. and his code is given in Fig. 6. The

```
      PROGRAM MULTI
      COMMON/GLOBAL/A(4ØØ), B(4ØØ), C(4ØØ)
      COMMON IVT1, IVT2
      DIMENSION IDT(2)
      EXTERNAL DOALL
      DATA B/4ØØ*1.Ø, C/4ØØ*1.Ø/

      NMAX = 4ØØ
      IDT(1) = 2

      CALL LOCKASGN(IVT1)
      CALL LOCKASGN(IVT2)
      CALL LOCKON(IVT1)
      CALL LOCKON(IVT2)

      T1 = 9.5E-9*RTC(DUM)
      T2 = 9.5E-9*RTC(DUM)
      TØ = T2-T1
      CALL TSKSTART (IDT, DOALL)

      DO 2Ø N = 2, NMAX, 2
      T1 = 9.5E-9*RTC(DUM)
      NHALF = N/2

      CALL LOCKOFF(IVT1)

      DO 1Ø I = 1, NHALF
 1Ø   A(I) = B(I)*C(I)

      CALL LOCKON(IVT2)

      T2 = 9.5E-9*RTC(DUM)
      T = T2-T1-TØ
      WRITE (6,1ØØ) N, T
 2Ø   CONTINUE

1ØØ   FORMAT('  N: ', I4, 4X, 'TIME IN SECONDS:', F16.12)
      STOP
      END
```

Fig. 4(a) Program for measuring r_∞ and $s_{\frac{1}{2}}$ when a job is split between the two CPUs of the Cray X-MP, using the LOCKS method of synchronization. Above code executed in CPU1.

```
      SUBROUTINE DOALL
      COMMON/GLOBAL/A(4ØØ), B(4ØØ), C(4ØØ)
      COMMON IVT1, IVT2

      NMAX = 4ØØ

      DO 2Ø N = 2, NMAX, 2
      NH1 = N/2 + 1

      CALL LOCKON(IVT1)

      DO 1Ø I = NH1, N
  1Ø  A(I) = B(I)*C(I)

      CALL LOCKOFF(IVT2)

  2Ø  CONTINUE

      RETURN
      END
```

Fig. 4(b) The subroutine DOALL used with Fig. 4(a), and executed in CPU2.

```
      PROGRAM MULTI
      COMMON/GLOBAL/A(4ØØ), B(4ØØ), C(4ØØ)
      COMMON IVT1, IVT2
      DIMENSION IDT(2)
      EXTERNAL DOALL
      DATA B/4ØØ*1.Ø/, C/4ØØ*1.0/

      NMAX = 4ØØ
      IDT(1) = 2

      CALL EVASGN(IVT1)
      CALL EVASGN(IVT2)

      T1 = 9.5E-9*RTC(DUM)
      T2 = 9.5E-9*RTC(DUM)
      TØ = T2-T1
      CALL TSKSTART (IDT, DOALL)

      DO 2Ø N = 2, NMAX, 2
      T1 = 9.5E-9*RTC(DUM)
      NHALF = N/2

      DO 1Ø I = 1, NHALF
 1Ø   A(I) = B(I)*C(I)

      CALL EVPOST(IVT1)
      CALL EVWAIT(IVT2)
      CALL EVCLEAR(IVT2)

      T2 = 9.5E-9*RTC(DUM)
      T = T2-T1-TØ
      WRITE (6, 1ØØ) N, T
 2Ø   CONTINUE

1ØØ   FORMAT ('  N: ', I4, 4X, 'TIME IN SECONDS:', F16.12)
      STOP
      END
```

Fig. 5(a) Program for measuring r_∞ and $s_{\frac{1}{2}}$ when a job is split between the two CPUs of the Cray X-MP, using the EVENTS method of synchronization. Above code is executed in CPU1.

```
      SUBROUTINE DOALL
      COMMON/GLOBAL/A(4ØØ), B(4ØØ), C(4ØØ)
      COMMON IVT1, IVT2

      NMAX = 4ØØ

      DO 2Ø N = 2, NMAX, 2
      NH1 = N/2 + 1

      DO 1Ø I = NH1,N
 1Ø   A(I) = B(I)*C(I)

      CALL EVPOST(IVT2)
      CALL EVWAIT(IVT1)
      CALL EVCLEAR(IVT1)

 2Ø   CONTINUE

      RETURN
      END
```

Fig. 5(b) The Subroutine DOALL used with Fig. (5a), and executed in CPU2.

```
      PROGRAM MULTI
      COMMON/GLOBAL/A(4ØØ), B(4ØØ), C(4ØØ)
      COMMON IVT1, IVT2
      DIMENSION IDT(2)
      EXTERNAL DOALL
      DATA B/4ØØ*1.Ø/, C/4ØØ*1.Ø/

      NMAX = 4ØØ
      IDT(1) = 2

      CALL INIT

      T1 = 9.5E-9*RTC(DUM)
      T2 = 9.5E-9*RTC(DUM)
      TØ = T2-T1
      CALL TSKSTART(IDT, DOALL)

      DO 2Ø N = 2, NMAX, 2
      T1 = 9.5E-9*RTC(DUM)
      NHALF = N/2

      CALL POST1

      DO 1Ø I = 1, NHALF
   1Ø A(I) = B(I)*C(I)

      CALL WAIT2

      T2 = 9.5E-9*RTC(DUM)
      T = T2-T1-TØ
      WRITE(6, 1ØØ) N, T
   2Ø CONTINUE

  1ØØ FORMAT ('  N: ', I4, 4X, 'TIME IN SECONDS:', F16.12)
      STOP
      END
```

Fig. 6(a) Program for measuring r_∞ and $s_{\frac{1}{2}}$ when a job is split between the two CPUs of the Cray X-MP, using CAL code for synchronization.

```
      SUBROUTINE DOALL
      COMMON/GLOBAL/A(4ØØ), B(4ØØ), C(4ØØ)
      COMMON IVT1, IVT2
      NMAX = 4ØØ

      DO 2Ø N = 2, NMAX, 2
      NH1 = N/2 + 1

      CALL WAIT1

      DO 1Ø I = NH1, N
   1Ø A(I) = B(I)*C(I)

      CALL POST2

   2Ø CONTINUE

      RETURN
      END

      IDENT  CALIB
      ENTRY  INIT, POST1, POST2, WAIT1, WAIT2
INIT  =      *
      A1     Ø
      SBØ    A1
      J      BØØ
POST1 =      *
      A1     1
      SBØ    A1
      J      BØØ
WAIT2 =      *
      AØ     SBØ
      JAN    *-1
      J      BØØ
WAIT1 =      *
      AØ     SBØ
      JAZ    *-1
      J      BØØ
POST2 =      *
      A1     Ø
      SBØ    A1
      J      BØØ
      END
```

Fig. 6(b) The Subroutine DOALL and the CAL simplified LOCKS synchronization routines used with Fig. 6(a). Code kindly supplied by John Larson (Cray Research Inc.).

overhead is thereby reduced by a factor of ten to $s_{\frac{1}{2}}$ = 220 flop. An examination of the CAL code in Fig. 6(b) shows that there is no wasted time, and it is unlikely that synchronization can be achieved on the Cray X-MP with less overhead. However, it must be said that in the CAL code, one CPU waits for the other to finish by continually testing one of the synchronization registers. This prevents the waiting CPU from doing any other work during this time, and hence this code would hardly be acceptable as a general method of synchronization.

4. CONCLUSIONS

Measurements have been reported of the vector startup overhead on a single CPU of the Cray X-MP, and the overhead of splitting a job between the two CPUs of a Cray X-MP. Interpreted in terms of the parameters (r_∞, $n_{\frac{1}{2}}$) we conclude that the Cray X-MP is about three times faster than the Cray-1 in asymptotic performance r_∞, however this has the effect of similarly increasing the half-performance length, $n_{\frac{1}{2}}$. One of the principle differences between the Cray-1 and the Cyber 205 is the fact that their values of $n_{\frac{1}{2}}$ differed by about a factor of ten (being ~10 to 20 and ~100 to 200 respectively). Now that we find the Cray X-MP with $n_{\frac{1}{2}}$ ~50 it has become more like the Cyber 205 from the vector length point of view. Other differences, like the need for contiguous vectors on the Cyber 205, of course, remain.

Measurements of job splitting between two CPUs interpreted in terms of the parameters (r_∞, $s_{\frac{1}{2}}$) show that the cost of job synchronization in terms of lost floating-point operations is about ten times more expensive on the Cray X-MP than on the Denelcor HEP1. This means that the grain size of jobs split between two CPUs needs to be about ten times larger on the Cray X-MP than on the HEP1 in order to achieve the same fraction of the maximum performance.

5. ACKNOWLEDGEMENTS

The author wishes to acknowledge his debt to David Dent of ECMWF who ran the programs described in this paper in a very short time, and made many valuable suggestions. This paper could not have been presented without his help. The author is also endebted to John Larson of Cray Research, Chippewa Falls, for his helpful criticism of the first draft of the paper, and for supplying the CAL code for the simplified LOCKS synchronization. The opportunity to discuss synchronization and its characterization with Paul Swarztrauber, Tor Block and others at the workshop is much appreciated. Dr Lennart Bengtsson, Director of ECMWF, and Mr Geerd-R. Hoffmann, Head of the Computer Division, are also thanked for granting their permission to use the ECMWF computer centre for these measurements. The author's work is supported by the UK Science and Engineering Research Council grant GR/B/39336.

6. REFERENCES

Calahan, D.A., 1977: Algorithmic and architectural issues related to vector processors, Proc. Intl. Symp. Large Eng. Syst., New York, Pergamon.

Calahan, D.A. and W.G. Ames, 1979: Vector Processors: models and applications, IEEE Trans. Circuits and Systems, CAS-26, 715-726.

Hockney, R.W., 1977: Supercomputer architecture, in Infotech State of the Art Conference: Future Systems, F. Sumner (ed.), Maidenhead, Infotech Intl., 277-305.

Hockney, R.W. and C.R. Jesshope, 1981: Parallel Computers - architecture, programming and algorithms. Bristol, Adam Hilger, pp 423. (Distributed in North and South America by Heyden & Son, Philadelphia).

Hockney, R.W., 1983: Characterizing computers and optimizing the FACR(ℓ) Poisson-solver on parallel unicomputers. IEEE Trans. Comput., C-32, 933-941.

Hockney, R.W., 1984: The $n_{\frac{1}{2}}$-method of algorithm analysis, in PDE Software: Modules, Interfaces and Systems, B. Engquist and T. Smedsaas (eds.), Amsterdam, Elsevier Science Publ. (North-Holland), 429-444.

Hockney, R.W. and D.F. Snelling, 1984: Characterizing MIMD computers: e.g. the Denelcor HEP, in Parallel Computing 83, M. Feilmeir, G.R. Joubert and U. Schendel (eds.), Amsterdam, Elsevier Science Publ., North-Holland, 521-526.

Hockney, R.W., 1985: Performance characterization of the HEP, in MIMD Computation: HEP Supercomputer and its applications, J.S. Kowalik (ed.), Cambridge, MIT Press.

Kuck, D.J., 1978: The Structure of Computers and Computation, vol. 1, New York, Wiley, pp 611.