GRAVITATIONAL N-BODY ALGORITHMS: A COMPARISON BETWEEN SUPERCOMPUTERS AND A HIGHLY PARALLEL COMPUTER

Junichiro MAKINO

Department of Earth Science and Astronomy, College of Arts and Sciences, University of Tokyo, 3-8-1 Komaba, Meguro-ku, Tokyo 153, Japan

and

Piet HUT

Institute for Advanced Study, Princeton, NJ 08540, USA

1989

NORTH-HOLLAND–AMSTERDAM

Contents

Computer Physics Reports 9 (1989) 199–246
North-Holland, Amsterdam

GRAVITATIONAL N-BODY ALGORITHMS: A COMPARISON BETWEEN SUPERCOMPUTERS AND A HIGHLY PARALLEL COMPUTER

Junichiro MAKINO

Department of Earth Science and Astronomy, College of Arts and Sciences, University of Tokyo, 3-8-1 Komaba, Meguro-ku, Tokyo 153, Japan

and

Piet HUT

Institute for Advanced Study, Princeton, NJ 08540, USA

Received 30 July 1988

We evaluate the performance of the Connection Machine, a highly parallel computer with 65,536 processors and a peak speed of 10 Gflops, on several types of gravitational N-body simulations. We compare the results with similar tests on a variety of more traditional supercomputers. For either type of computer, the most efficient algorithm for simulating an arbitrary, very large system of self-gravitating particles, such as star clusters or galaxies, has a force calculation pattern based on a tree structure. This tree structure is highly irregular and rapidly changing in time. This algorithm therefore presents an extreme challenge for hardware as well as software of fast computers. We present benchmarks for this algorithm and also for a much simpler algorithm, together with a detailed analysis of the factors which determine the efficiency.

On vector supercomputers we have measured typical tree-code performances to have an efficiency (fraction of peak speed) of ~1% without handcoding, using vectorizing compilers; a typical efficiency of a few percent when making special modifications in the basic algorithm and modifications in the code to improve vectorization; and a maximum efficiency obtained on any supercomputer of <15%. On the Connection Machine CM-2 we found the efficiency of a straightforward implementation to be ~0.01%; for an handcoded version we reached ~0.7%; while we predict the present hardware limitations to provide a ceiling of 2 ~ 3% efficiency even with future software and firmware improvements.

From a hardware point of view, vector pipeling supercomputers and fine-grained parallel computers such as the Connection Machine are very different. Nevertheless, the structure of optimal algorithms and the efficiency of their performance is rather similar. The similarity in the structure of the algorithms is a reflection of the fact that both types of computer have SIMD (Single Instruction Multiple Data) architectures. The rather low efficiency is the result of the fact that these machines are designed for optimal performance on rather simple algorithms, e.g. finite difference method using a regular grid.

0167-7977/89/$16.80 © Elsevier Science Publishers B.V.
(North-Holland Physics Publishing Division)

1. Introduction

During the last decade, supercomputers with pipeline architectures have formed the most powerful tool for large scale scientific computations. Their computational power derives from advances made in two different technologies. One is the increase in switching speed of elementary components, which increases the speed of any type of operation. The other is the development of vector pipelines, which makes it possible to execute vector operations very fast. A vector operation is a single operation which is applied to many elements of an array, by interleaving parts of the operation, thereby excluding interdependency of the operations on different elements. This implies that these operations can be viewed equally well as taking place in parallel. Pipelined supercomputers thus have a SIMD (Single Instruction Multiple Data) architecture.

Recently, a third technological improvement has appeared, in the form of multiple pipelines and/or multiple processors. This development is driven by the fact that the speed of a single processor/pipeline is getting close to the theoretical limit, set by the finite speed of light. For example, both the Cray X-MP and Cray-2 have four scalar processors and four vector pipelines which can execute simultaneously, and the ETA-10 has 8 scalar processors each with its own set of vector pipelines. At present, no Japanese supercomputer has more than one scalar processor, but all have multiple pipelines which can work in parallel to some extent. Unfortunately, the introduction of pipelines and multiple processors to speed up the hardware carries with it a software penalty: it requires the development of different and more complicated algorithms.

There is yet another way to obtain a large computing power. If we can use a large number of processors in parallel, the computational speed will exceed that of existing supercomputers, even for a relatively low speed per processor. This idea is realized in the Connection Machine [1], which contains 65,536 processors. The Connection Machine system consists of two components, a processor array and a host machine. Each processor has its own local memory for data storage. All processors simultaneously execute a single instruction issued by the host machine. Thus, the Connection Machine, too, has a SIMD architecture.

From a hardware point of view, pipeline machines and the Connection Machine are very different. Nevertheless, the characteristics of the algorithmic adaptations and their efficiency are largely similar for both types of machines. The reason is that both types of machine belong to the SIMD class, and both show serious bottlenecks for *arbitrary* patterns of memory access (because the basic design is always geared towards optimal performance for *regular* patterns of memory access). This similarity makes it possible to make a meaningful comparison between the performance of supercomputers and the Connection Machine.

In our study we have concentrated on the gravitational N-body problem, which plays a central role in many large scale simulations in astrophysics. In two ways this problem is intrinsically more complicated than the more familiar hydrodynamics computations. First, individual stars in a galaxy can move freely around on individual orbits without colliding, i.e. their mean free path is much longer than the size of the system. Secondly, the gravitational force holding a galaxy together is a long-range force, which makes it impossible to divide the whole system into subsets which interact only along the interfaces. Thus both the particle configurations and the particle–particle interactions are globally interrelated and highly time-dependent.

Notwithstanding these complications, the N-body problem is in principle well structured for vector/parallel decomposition. The most time consuming part is the evaluation of the gravitational force. The force on any one particle is evaluated independently of the force on all other particles. Thus, the degree of parallelism at least equals the number of particles itself (at least in those algorithms in which all particles are propagated in lockstep). On vector processors, this means that we can use very long vectors. On the Connection Machine, this implies that we can use a large number of processors.

A very simple algorithm and a fairly complex but potentially more efficient algorithm are discussed separately. In terms of hardware speed, both supercomputers and the Connection Machine show a very efficient performance for the simple algorithm, while for the sophisticated algorithm both show a significant degradation of the performance. This loss of efficiency is more pronounced for the Connection Machine, because of limitations imposed by the current software.

In section 2 we give a quick overview of the main characteristics of vector processors and the Connection Machine. In section 3 we discuss how we can apply these machines to realistic problems. We discuss the gravitational N-body problem as our main example, and compare the performance of different algorithms, with a detailed discussion of specific modifications which can speed up the calculations. In section 4 we augment our comparisons of different computers by taking into account the programming environment as well. Section 5 contains a discussion of our results.

2. Architectures

In this section, we give a brief description of the characteristics of vector processors and highly parallel computers. In section 2.1 we discuss hardware and performance characteristics for basic arithmetic operations. In section 2.2 we discuss the performance for more complex operations such as conditional statements and mathematical functions. In section 2.3 we discuss how the performance is affected by irregular memory access. This last issue is crucial both for pipeline and for parallel machines. For vector pipelines, to access the memory consecutively is much faster than to access it randomly. For a parallel machine with processors having their own local memory, it is much faster for a processor to access its own memory than to access the memory of other processors. Therefore, random access operations can easily dominate the total computing time for complex algorithms.

2.1. Hardware and basic performance characteristics

2.1.1. Vector processors: traditional supercomputers

Vector processors contain three main components: a central scalar processor; a linearly addressable memory; and vector pipelines. The first two parts work in the same way as in conventional sequential computers. A vector pipeline is a specialized piece of hardware which executes vector operations much faster than the scalar processor. For example, consider a DO loop of the following form:

```
do i = 1,N
    a(i) = b(i) + c(i)
enddo
```

This is the most basic form of a vectorizable DO loop; one which a vector pipeline can execute. Each iteration of the loop operates on different elements of the arrays. Therefore the calculation of the ith iteration does not depend on the result of previous iterations. This implies that all iterations can be executed in parallel.

With a scalar processor, the time required to execute a DO loop is proportional to the number of iterations,

$$T_{\text{scalar}} = aN, \tag{2.1}$$

while the number of iterations per unit time, i.e. the rate at which iterations are performed, is given by

$$R_{\text{scalar}} = \frac{1}{a}. \tag{2.2}$$

In a vector pipeline, the time required for the execution of the above loop has the following form:

$$T_{\text{vector}} = b + cN. \tag{2.3}$$

The first term is the start-up time for the pipeline. In addition, the pipeline requires a time proportional to N to finish the DO loop. The number of the iterations of the DO loop per unit time for a vector processor is given by

$$R_{\text{vector}} = \frac{N}{b + cN} \approx \frac{1}{c}, \quad (N \gg 1). \tag{2.4}$$

In general $c \ll a$ (the peak performance of a pipeline far exceeds that of the scalar processor), but $b \gg c$ (this peak performance is not reached for short vectors). Table 1 shows the values for a, b and c for the above DO loop measured on several vector processors. Note that these values are obtained for the specific array addition operation described above.

Figure 1 shows R_{vector} as a function of vector length N for values of b and c in table 1. If N is small, the pipeline can be slower than the scalar processor due to the start-up time. With larger N the relative speed of the pipeline increases but saturates for very large N.

From fig. 1 we can see how the values of both b and c determine the actual performance. However, in many large scale simulations the length of the vectors which appear in the core of the program is very long. Thus, the value of c is more likely to dominate the performance.

Table 1
Performance characteristic of the vector processors. Unit is microsecond

Machine		a	b	c	a/c	b/c
Cray X-MP/18	(CFT1.15)	0.39	1.18	0.015	26	80
Cyber 205	(FTN200)	0.35	2.19	0.010	35	219
Hitac S-820	(FORT77/HAP V21-0A)	0.13	0.40	0.0025	53	161
Facom VP-400	(FORT77/VP V10L20)	0.36	1.55	0.0055	65	280

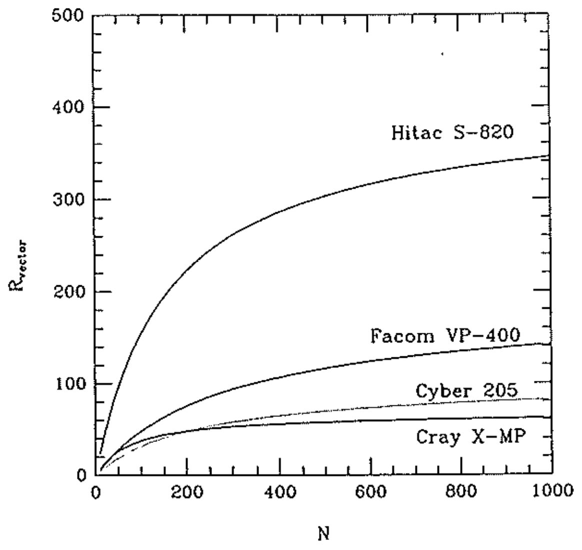

Fig. 1. Performance characteristics of several vector processors. The calculation speed for a(i) = b(i) + c(i) is plotted as a function of the loop length N, in unit of Mflops.

Table 2 shows the peak speed of vector processors for various kind of operations. The values for simple arithmetic operations indicate the maximum performance possible on a machine. The peak speed of the division is significantly smaller than addition, subtraction and multiplication, on all of the listed machines.

Table 2
Speed of vector processors. Unit is Mflops (millions of floating point operations per second). The iteration count for the loop is 10^4. All timings are for double precision (64 bits)

	Cray X-MP *	Cyber 205 **	Hitac S-820	Facom VP-400
a(i) = b(i) + c(i)	67	100	410	179
a(i) = b(i) * c(i)	66	100	410	179
a(i) = b + c(i) * d(i)	133	200	790	358
a(i) = b(i)/c(i)	25	16	185	61
a(i * k) = b(i * k) + c(i) (k = 2)	49	33	615	176
a(i * k) = b(i * k) + c(i) (k = 64)	13	11	24	38
a(j(i)) = b(i) ***	47	30	140	49
a(i) = b(j(i)) ***	38	30	270	76
sum = sum + b(i)	109	50	626	395
a(i) = sqrt(b(i))	13	16	24	4
if(b(i).gt.0.0)a(i) = b(i) ****	24	25	250	68
if(b(i).gt.0.0)a(i) = b(i) *****	5	5	50	14

* Cray X-MP/216, 1 processor.
** With 2 pipes.
*** a(j(i)) etc. denote indirect addressing.
**** The ratio of positive b(i) is 50%.
***** The ratio of positive b(i) is 10%.

Note that the speed for simple arithmetic operations is usually smaller than the peak performance as advertised by the manufacturers. There are two main reasons for this decrease in speed: (1) current vector processors have multiple pipelines; (2) the bandwidth between the main memory and the vector registers is not wide enough to keep the arithmetic pipelines busy.

For example, a Hitac S-820/80 has four pipes for addition, four pipes for multiplication, and four more pipes for addition that are connected to the pipes for multiplication. Thus the total number of the arithmetic pipelines is 12. The cycle time for the vector unit of a S-820 is 4 ns, and each pipe can perform one arithmetic operation per clock period. Therefore the theoretical peak speed of one pipe is 250 Mflops and the peak speed of a whole machine containing 12 pipelines is 3 Gflops. On the other hand, a S-820 has four pipes for vector load from the main memory to the vector registers and another four load/store pipes that can be used either as load pipes or store pipes. These load pipes and load/store pipes also operate with the cycle time of 4 ns. In the case of a vector addition, load/store pipes are used to store the results. Thus only four pipes are available to load the vectors from the main memory. This means that the speed for loading the vectors is 1 Gwords/s, because each of the 4 load pipes loads one vector element every 4 ns, resulting in an effective speed of 1 word/ns. Therefore the speed of the vector addition cannot exceed $\frac{1}{2}$ Gflops = 500 Mflops, even if there is no extra overhead, because one addition requires the loading of two vector elements. In general, an operation such as a(i) = b + c(i) * d(i) will be performed at a speed in between that of single arithmetic operations and the advertised top speed, as can be seen in table 2.

All manufacturers advertise the peak performance as the speed attained if all pipelines which can work in parallel actually do work in parallel. In practice, this situation is hardly ever realized and even for very long vectors the performance is generally significantly smaller than the advertised peak speed.

2.1.2. A highly parallel computer: the Connection Machine

The Connection Machine contains two major components, the host machine and the processor array. The host machine is a conventional scalar processor (a VAX or a Symbolics Lisp Machine). The processor array is a cluster of 65,536 processors, each having 8 kbytes of local memory for the largest configuration currently available.

There are other machines which fall under the category of "highly parallel" (e.g. the ICL DAP and the Goodyear MPP). There are, however, two major features which make the Connection Machine special among highly parallel machines. The first is the number of processors which is the largest for the Connection Machine. This large number of processors gives it the largest peak performance, of about 10 Gflops for the case of the CM-2, as advertised by the manufacturer. The second and more important feature of the Connection Machine is the fact that it has a general communication network. With the Connection Machine, every processors can communicate with every other processors via a general communication network.

Two high-level languages are currently available on the Connection Machine: C* and *Lisp. C* is a parallel extension of the language C and *Lisp is a parallel extension of Common Lisp. Thinking Machines Corporation has also announced that they will support Fortran 77 with array extensions of Fortran 8X. That implies, however, that in the meantime they will not develop a "paralellizing compiler" which will automatically generate machine code for the parallel execution of serial Fortran programs. Most vector processors support some type of vectorizing compiler which generates vector instructions for serial Fortran programs. In principle this could imply that the vector processors are easier to use than the Connection Machine. In practice, however, it turns out that many programs written for scalar machines do not run efficiently on vector processors. Moreover, it is not clear whether serial Fortran is an appropriate language for parallel/vector machines. This point will be discussed in section 4.

Independently of the language we use, the Connection Machine operates in the following way. The parallel variables are stored in each processor. When an instruction for the parallel operation is issed from the front end, all processors operate simultaneously. However, each processor has one bit flag called the *context flag*, which determines its current state of activity. For most statements only processors with a non-null context flag are activated. Those processors with a null context flag remain idle. Thus the amount of computation executed by the processor array is proportional to the number of processors activated. A few statements are *unconditional*, i.e. they are executed regardless to the value of the context flag. These unconditional statements are required, for example, to reset the context flag itself.

The number of processors is currently limited to 64k. This would seem to imply that we should change our algorithm whenever we would use the Connection Machine for a data set containing more than 64k elements. However, the Connection Machine provides a complete solution for this problem, in firmware. It supports a virtual processor system in which the physical memory of each processor can be divided into n portions in such a way that each processor will simulate n distinct processors. This virtual processor system is supported on the firmware level, just as virtual memory is supported on conventional computers. Therefore, user programs need not care about the number of processors physically available. The speed is, however, slower for a larger number of virtual processors.

Let us again consider again the array-addition DO loop which we discussed in the previous section:

```
do i = 1,N
    a(i) = b(i) + c(i)
enddo
```

Although we use the same notation as before for convenience, please note that the language used in the Connection Machine does not access the variables stored in the processor array via such an index. In C*, the statement equivalent to the above DO loop is a = b + c. In Lisp, the parallel addition is expressed as (+!! b c). In Fortran it will be something like a(1 : N) = b(1 : N) + c(1 : N). In this paper, however, we would like to use one abstract language to describe the operations on both vector processors and the Connection Machine. Therefore, we use a simple pseudo-Fortran throughout this paper. The translation from the Fortran-like syntax to the languages actually used on the Connection Machine is generally straightforward. As an example, programs in C* can be found in Appendix A.

For the Connection Machine, the time required to execute the above DO loop is constant, as long as the number of iterations does not exceed the number of physical processors. To deal with arrays larger than the size of the physical machine, the Connection Machine stores several elements of an array in one processor and locally operates on them sequentially. Therefore, the

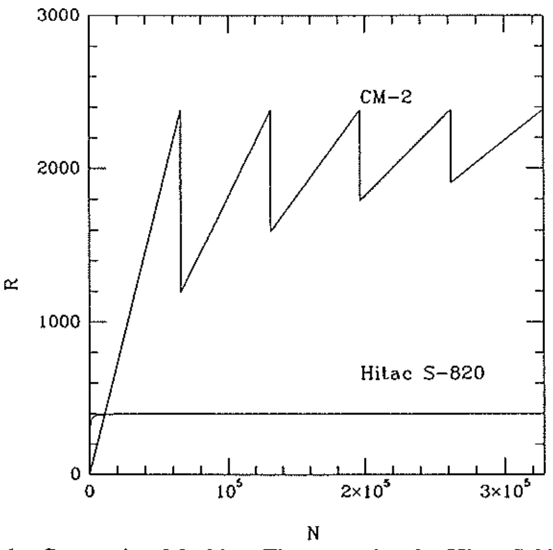

Fig. 2. Same as fig. 1 but for the Connection Machine. The curve for the Hitac S-820 is shown for comparison. Note that the range of the loop length is much wider for this figure than for fig. 1.

time required to execute the above DO loop is given by

$$T_{\mathrm{CM}} = d\left[\frac{N+65{,}535}{65{,}536}\right], \tag{2.5}$$

where $[x]$ denotes the maximum integer that does not exceed x. The number of iterations executed per unit time can thus be expressed as:

$$R_{\mathrm{CM}} = \frac{N}{d\left[\dfrac{N+65{,}535}{65{,}536}\right]}. \tag{2.7}$$

The value of d is considerably larger than the value of a in eq. (2.1) for the front end. In addition, the value of a for the front end of the Connection Machine (e.g. a VAX 8800) is significantly larger than that for a scalar processing unit of a supercomputer. The value of a is about 2 μs, and $d \approx 50$ μs. Fig. 2 shows the behaviour of the performance for the Connection Machine. The main difference between fig. 1 and 2 is that in fig. 2 the performance saturates for much larger N values. This means that we need to process a large number of data to use the Connection Machine effectively.

Table 3 shows the speed of the Connection Machine CM-2 for several basic operations. The numbers in the second column is the speed of one processor in kflops (kilo floating-point operations per second). The third column shows the measured peak speed for 8k processors with a 4 MHz clock speed, while the last shows the estimated peak for 64k processors for an 8 MHz clock speed. The projected speed for the simple arithmetic operations on the 64k-processor

Table 3
Speed of the Connection Machine. All timings are for single precision (32 bits). The number of data is the same as the number of processors available

	per processor (kflops, 4 MHz)	8k processors (Mflops, 4 MHz)	64k processors * (Mflops, 8 MHz)
a(i) = b(i)	30	246	3930
a(i) = a(i) + b(i)	18.2	149	2390
a(i) = a(i) * b(i)	16.7	137	2190
a(i) = a(i)/b(i)	5.0	41	655
a(j(i)) = b(i) (best)	2.6	21	340
a(j(i)) = b(i) (usual)	0.71	5.8	93
a(i) = b(j(i)) (best)	1.1	9	144
a(i) = b(j(i)) (usual)	0.33	2.7	43
a(i) = sqrt(b(i))	2.9	24	380
if(b(i).gt.0.0)a(i) = b(i) **	2.6	21	340
if(b(i).gt.0.0)a(i) = b(i) ***	0.52	4.2	68

* All values in this column are estimated from results of timings on 8k processors.
** The ratio of positive b(i) is 50%.
*** The ratio of positive b(i) is 10%.

machine is quite impressive. These numbers are much larger than that of current vector processors.

We should warn the reader here that the numbers in table 3 are not the actual times needed for performing operations on the Connection Machine. These timings (and all timings on the Connection Machine in this paper) are based on the elapsed time as measured on the host machine. Therefore, they include a small amount of overhead caused by the operating system of the host machine. Thus, it is possible that we underestimate the performance of the Connection Machine to some extent. It is of course possible to measure the net time during which the processor array is actually working. However, we are mainly interested in the practical performance of the Connection Machine as a whole, rather than the theoretical speed of the bare processor array. Therefore we prefer to include the overhead of the front end in the timings.

Also for the Connection Machine, the numbers for simple arithmetic operations are smaller than the advertised peak speed of ~ 10 Gflops. The reason for this degrading is quite similar to that discussed for vector processors: the speed of 10 Gflops is attained when all processors calculate a vector dot-product within the processor. With the dot-product, the floating-point accelerator operates in a pipelined fashion, and therefore the number of executed operations per unit time is larger than in the case of a single operation per processor.

2.2. More complex operations

In section 2.1 we discussed performance characteristics of different types of computers for simple arithmetic operations. Here we extend our discussion to somewhat more complex operations, such as conditional operations and mathematical functions.

2.2.1. *Vector processors*

The basic mode of operation of a vector pipeline is to apply the same arithmetic operation to all elements of the vector. In practice, however, we need richer set of operations to perform real work. Most vector processors nowadays provide some way to execute conditional statements and arithmetic functions. Thus, they can vectorize statements like the following:

```
a(i) = sqrt(b(i))          (arithmetic functions)
if (b(i) .gt. 0) then
  a(i) = b(i)
endif                      (conditional execution)
```

Nevertheless, these operations are relatively slow on most vector machines. Table 2 shows the peak speed for these operations for different computers. Note the rather low performance for these operations on some machines.

The figures for conditional execution require further explanation. The conditional execution is slower than its unconditional equivalent for two reasons: (1) the evaluation of the condition itself takes a considerable amount of time; (2) the conditional vector operations decrease the amount of computations performed without decreasing the execution time. In the conditional assignment statement of table 2, the number of the elements copied from array b to array a is smaller than N, although the time required to execute the vectorized DO loop still obeys eqs. (2.3).

2.2.2. *The Connection Machine*

The Connection machine can also execute conditional statements and arithmetic functions in parallel. The relative speed of these operations, as compared to simple arithmetic operations is similar to that of vector processors, as can be seen from table 3.

2.3. *Non-contiguous accessing of the memory*

Vector processors are designed to operate on vectors. A vector is a set of contiguous addresses in the memory. Therefore, the access speed of the pipeline for non-contiguous addresses is significantly slower than the access speed for vectors. In the Connection Machine, each processor has fast access only to its local memory. It requires a much longer time for a processor to access memory of other processors. Thus both machines exhibit a similar type of performance degradation for complex patterns of memory access.

2.3.1. *Vector processors*

Typical patterns of non-contiguous memory addressing are:

```
a(k*i) = b(k*i) + c(i)      (constant stride)
a(j(i)) = b(1(i)) + c(i)    (indirect addressing)
sum = sum + b(i)            (reduction by summation)
```

The last type of summation looks like a sequential operation, because the partial sum up to the ith element cannot be obtained without knowing the partial sum for up to the $(i-1)$th element.

However, most current types of vector hardware provide a relatively efficient summation, by rearranging the partial summation in such a way that different parts of the array are added in an interleaved manner. When discussing the speed of, for example, addition on vectors, we have to discriminate between the time required for the vector addition of two different vectors to produce a third vector, and that for the summation of all elements of a single vector ("reduction of a vector under summation"). In both cases, for vectors of length N, the number of operations equals N and $N-1$ for vector addition and reduction, respectively. In practice, however, the amount of time required for reduction is not the same as that for addition. On a vector machine, reduction of a vector is usually faster than vector addition. A vector addition for vectors of length N requires access of $3N$ words. On the other hand, a reduction of a vector of length N requires access of only N words. Thus, a vector reduction is faster than a vector addition. On a fine-grained parallel machine, such as the Connection Machine, the situation is reversed. To perform one parallel addition, each processor simply execute one addition. In order to perform the reduction, first all processors are configured to a binary tree. A reduction operation is performed by sending messages through the tree. First the processors in the lowest level send messages to their parents. Then the processors in the second level add the data that they received. Then they send data to their parent and this process continues until the processor at the top of the tree receives the data. The depth of the tree is $\log_2 N$. Thus reduction is slower than vector addition by a factor $\log_2 N$ [1,2].

Table 2 also shows the peak speed for these operations. Note that they are much lower than that of simple arithmetic operations shown in the top of the table, except for the summation which is relatively fast. This implies that the actual cost of an algorithm on vector processors cannot be estimated accurately without considering the cost of memory access. In section 3 we discuss in detail how this problem affects the performance of different types of algorithms.

2.3.2. The Connection Machine

The most important characteristic of the Connection Machine is that each processor can access the local memory of any other processor via the communication network.

There are two ways to access the memory of other processors. In one case, each processor writes a value to the memory of other processors. This is called the SEND operation, because each processor sends a value to other processors. In the other case, each processor reads the memory of other processors. This is called the GET operation. Actually SEND is the only basic operation and GET is implemented via SEND. With GET, first all processors send their identification number (id) to those processors for which they want to read some of their local memory. Then the processors which receive the id send the data to the processor corresponding to that id. This implies that a GET takes twice as much time as a SEND operation.

Table 3 shows the speed for these operations. The indirect addressing with the form a(j(i)) = b(i) stands for a SEND and a(i) = b(j(i)) stands for a GET. The speed depends on the pattern of communication. It is high for a regular communication pattern, in which each processor communicates with a processor which is physically near in the network. Otherwise, it can take considerably more time.

The GET operation is more complicated to analyze. It is possible that many processors will request data simultaneously from the same one processor. At the time of our benchmarking, one processor could receive only one request at a time and would then reply to it individually.

Therefore, if a hundred processors happen to request the data from a single processor, it takes a hundred times as long as the time required if all processors would request data in different processors. A later new version of the microcode, which will be able to deal with this situation more efficiently, will be provided. With this new microcode, the communication network will automatically construct a fan-out tree for simultaneous requests to the same processor, and propagate the data via the tree.

Upon comparing tables 2 and 3, we find that the relative speed of the indirect addressing is significantly slower for the Connection Machine, although the absolute speed is similar for vector processors and the Connection Machine. This implies that for a communication-intensive algorithm, the performance degradation for the Connection Machine is more serious than that for vector processors. Since the absolute communication speed is similar for both types of machines, we can reasonably expect that also the attained absolute performance is similar, for communication-intensive algorithms.

We should mention here that this is a specific problem only for the CM-2. The CM-2 has almost the same hardware as CM-1, except for the attached floating-point accelerators (FPAs). The FPAs increase the speed of arithmetic operations by more than an order of magnitude, while the communication performance has not changed very much. Thus the performance of communication and calculation are not well balanced in the CM-2. The peak speed for arithmetic operations of the CM-1 is around 100 Mflops, which is well balanced with its speed of communication.

Except for the drawback of a relatively low speed, the communication network of the Connection Machine is significantly more powerful than the indirect-addressing ability of vector processors. For example, consider the following operation

```
do i = 1,N
    a(id(i)) = a(id(i)) + b(i)
enddo
```

If id(i) has the same value for different values of the index i, vector pipelines cannot guarantee a correct result. However, the general communication network of the Connection Machine can deal with this operation, at least for fixed-point numbers at the time at which we performed our tests. Later implementations will also be able to deal with floating-point numbers.

2.4. Summary

In this section we have discussed in detail the performance characteristics of vector processors and the Connection Machine. Each type of computer can execute in a non-scalar mode almost everything that the other type of machine can execute. However, the performance can be very different, depending on the type of calculation. The peak performance for simple arithmetic on the Connection Machine is better than that on vector processors. The performance for indirect addressing, however, is quite similar. The speed for short vector or scalar operations is by far faster on vector processors.

In the next section, we will discuss how these characteristics influence the performance of specific application programs.

3. Algorithms, implementations and performance

3.1. The gravitational N-body problem

The gravitational N-body problem is described by the following equations of motion:

$$\frac{d^2 r_i}{dt^2} = \sum_{j \neq i} \frac{G m_j (r_j - r_i)}{|r_j - r_i|^3}, \tag{3.1}$$

where r_i is the position of the ith particle, m_i is the mass of the ith particle, and G is the gravitational constant. Since gravity plays an important role in most astrophysical calculations, the above equations are encountered whenever one wants to study the detailed behavior of a gravitating system in a particle approximation. In some of these calculations, each particle directly simulates a physical object, such as a planet, a star, a gas cloud or a whole galaxy. More often, individual particles represent a sample of a larger body, such as a fraction of the mass distribution of a whole galaxy. In addition to eq. (3.1), local interactions can be added, e.g. to describe hydrodynamical effects. Although we will limit our discussion in the present paper to purely gravitational forces, it is interesting to note an increased use of particle methods to simulate gravitating fluids, by attaching thermodynamic variables such as pressure and entropy to the particles as well (cf. [3]).

The most direct approach to solving eq. (3.1) requires N force calculations per particle (the summation over j), resulting in N^2 inter-particle force calculations per time step. This is a reflection of the long-range character of gravity. However, a full N^2 set of force calculations is very expensive for simulations with high spatial resolution, which require a very large particle number. For example, with $N = 10^5$, one time step would involve a few times 10^{11} floating-point operations, or about an hour on a supercomputer with an effective speed of 100 Mflop. This number follows from eq. (3.1), in which a typical force calculation can be seen to involve $30 \sim 70$ floating-point operations (the uncertainty arising from the machine dependence of the treatment of the square root operation involved in computing the denominator).

Many solutions have been proposed and implemented to circumvent the prohibitive costs of naive N-body calculations. These can be classified into two groups:

(a) One type of solutions reduces the frequency at which individual particles have to compute the total force which other particles exert on them.

(b) Another class of solutions reduces the cost per particle to obtain this total force.

The first type includes the use of higher order integrators and of individual time steps (particles in denser regions are given shorter time steps). We refer to the review by Aarseth [4] for a discussion of these methods, and to Makino and Hut [5,6] for a detailed analysis of the computational complexity of a number of different methods in class (a).

The second type includes a large spectrum of solutions, many of which have built-in assumptions about geometrical symmetry or near-homogeneity, or become increasingly inefficient when deviations from homogeneity and simple geometry become large. General references can be found in the book by Hockney and Eastwood [7], and in the proceedings of a workshop [8], in which also issues such as vectorization and parallelization of N-body methods were discussed. Recently, a new class of algorithms has been developed which does not use grids and is free from geometrical assumptions. These codes simplify the force calculation on an individual particle, by grouping together the force contribution of other particles in such a way that the number of particle–particle force calculations grows less rapidly, typically $\sim N \log N$ [9–12] or even $\sim N$ [13,14]. These codes are often called tree codes, since they all use some form of a hierarchical tree structure for bookkeeping purposes.

The basic idea of a tree code is to represent the force from distant particles by a multipole expansion. In the present paper, we will limit our analysis to an $O(N \log N)$ algorithm which only incorporates the monopole moments [12]. This type of tree algorithm will be discussed in section 3.3, after the discussion of the simpler but less efficient N^2 algorithms in section 3.2.

3.2. Direct summation: a regular communication pattern

A direct summation code evaluates the right-hand side in eq. (3.1) directly, thereby calculating the force from each particle onto each other particle. Therefore, the number of forceterms grows as $O(N^2)$.

3.2.1. Vector processors

We start our discussion with the following efficient direct summation algorithm for a vector machine:

algorithm (a): direct method for vector processors

```
do j = 1, N - 1
    do i = j + 1, N in parallel
        do k = 1,3
            dx(i,k) = x(i,k) - x(j,k)
        enddo
        r2(i) = dx(i,1) * *2 + dx(i,2) * *2 + dx(i,3) * *2
        r3inv(i) = 1/(r2(i) * sqrt(r2(i)))
        do k = 1,3
            a(i,k) = a(i,k) - dx(i,k) * (m(j) * r3inv(i))
            a(j,k) = a(j,k) + dx(i,k) * (m(i) * r3inv(i))
        enddo
    enddo
enddo
```

The key word "in parallel", following the first line of a do loop, indicates that this loop can be fully vectorized. As we discussed in section 2, a DO loop which is vectorizable on a supercomputer can also be executed in parallel with multiple processors. Here and below, initialization of the form $a(i,k) = 0$, etc., is implicitly understood to have taken place earlier in the program. Furthermore, the innermost short loops over k are in practice written out explicitly for $k = 1$, $k = 2$ and $k = 3$ to indicate to the compiler that the i loop is effectively the innermost loop which should be vectorized. Note that this algorithm makes use of the fact that the gravitational force is symmetric, i.e.

$$f_{i,j} = -f_{j,i}.$$

Figure 3a shows how this algorithm works. The region indicated by a thick line stands for the

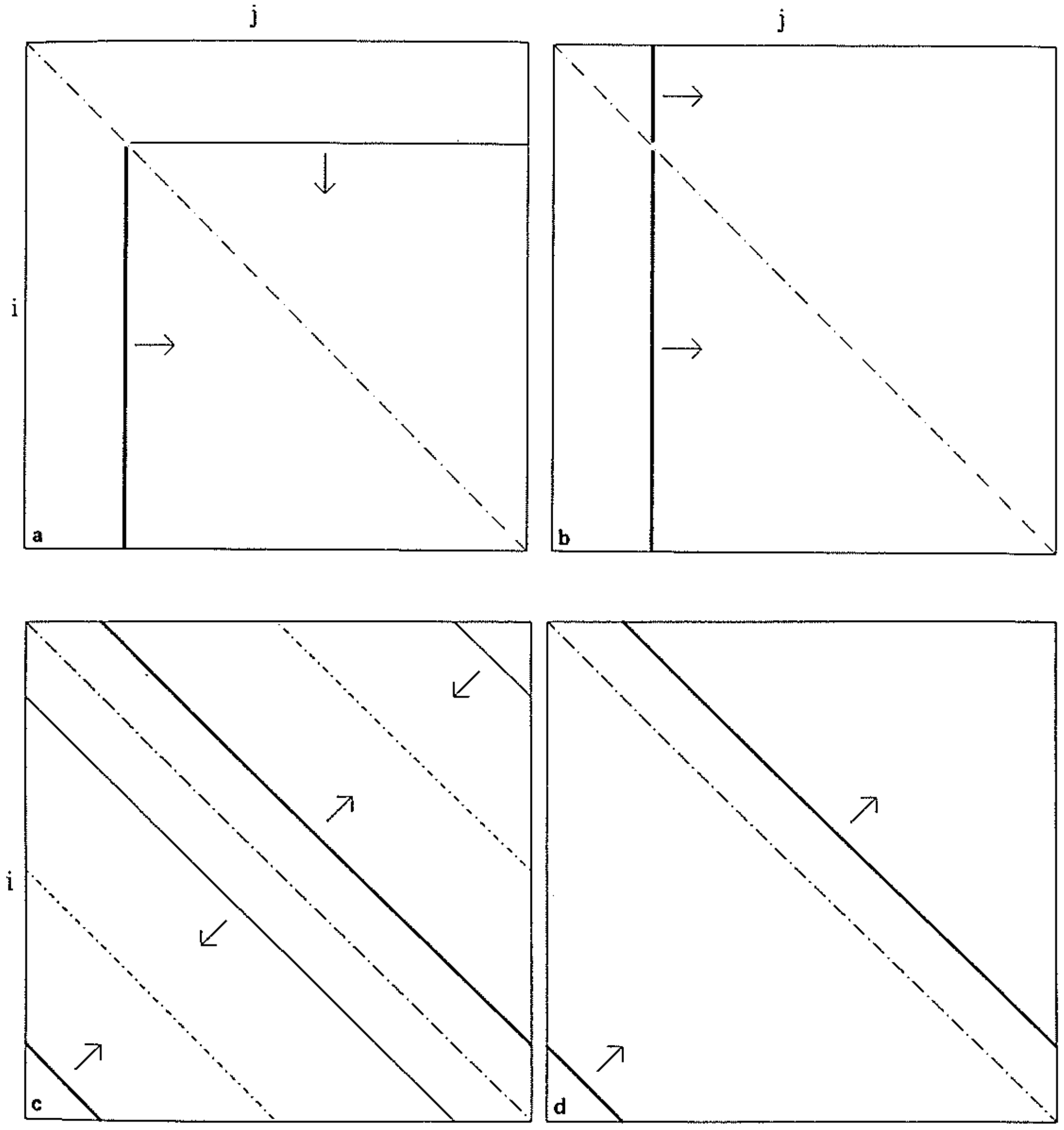

Fig. 3. Schematic description of direct summation algorithms. The box indicates the N^2 interactions. The dot–dash line indicates the self-interaction which must be avoided. The thick line indicates the pair of particles of which interactions are calculated in parallel. The thin line indicates the interactions which are obtained by using the symmetry. Arrows indicate the direction to which the calculation goes on. (a) is for algorithm (a); (b) is for algorithm (b1); (c) is for algorithm (b2); (d) is for algorithm (b3).

collection of pairs (i, j), for fixed j and all i values which are treated in parallel $(i > j)$. For each (i, j) pair, the corresponding force term is added to the partial forces on all particles i. The thin line stands for the pairs (j, i), for which the force term is obtained by symmetry, without additional computation; these contributions are all added to the force on the one particle j. The calculation proceeds in the direction indicated by the arrows, when stepping through the outer loop.

On a scalar computer, such as a workstation or mainframe, the above algorithm can be used efficiently as well, with a slight modification: there is no need to use a 2-D array for dx and 1-D array for r2 and r3inv. Such a scalar version of algorithm (a) can be run efficiently on many vector machines, since most vectorizing compilers can vectorize the code written using scalar temporary variables, by allocating either a vector register or temporary arrays. In this case, we can use the scalar version of the code which looks more natural. Here we give algorithm (a) in the above form in order to show how the pipeline execution of the algorithm takes place.

Algorithm (a) works effectively on all pipeline machines. The average length of the vector passed to the pipeline is $N/2$, which is long enough to extract peak performance of most vector processors for $N > 10^3$. The actual performance of this algorithm will be discussed in detail in section 3.2.3.

Note that here we have shown the algorithm, which is somewhat different from the actual piece of code. To optimize this algorithm for either scalar machines or vector machines, we should be careful concerning a number of subtle points. For example, in Fortran an array is allocated in the column-major order, i.e. x(1,1) is followed by x(2,1). Therefore, for vector processors the above example constitutes an efficient implementation, because all vector operations are applied on elements in contiguous addresses. However, with a scalar machine which includes virtual memory and a small amount of fast memory, the order of the indices should be reversed to minimize the memory swapping. In addition, as mentioned above, most of the vectorizing compilers consider only the innermost DO loop as the target of the vectorization. With these types of compilers, the innermost loop of algorithm (a) should be unrolled. This unrolling of the DO loop may also improve the performance of the scalar machine, because it eliminates the overhead caused by the looping. Thus to extract the maximum performance is not trivial even for a very simple algorithm like the one above.

3.2.2. The Connection Machine

On the Connection Machine, algorithm (a) is not very efficient. The reason is that the line

```
a(j,k) = a(j,k) + dx(i,k) * (m(i) * r3inv(i))
```

requires reduction under summation (see section 2.3.1). This summation over the subsequent elements in a vector does not pose a problem for vector processors, because most vector processors can perform reduction under summation at comparable speed as vector addition.

For the Connection Machine, however, the situation is quite different. The parallel addition takes a constant amount of time (independent of N as long as N does not exceed the number of processors). Reduction over summation in a vector (i.e. the summation of N terms in one sum) takes $O(\log N)$ time. This means that for practical values of N, a summation takes at least an order of magnitude longer than the time required for simple parallel addition.

We can adapt algorithm (a) by avoiding reduction operations altogether, i.e. by leaving out the last statement a(j,k) = ... This implies a doubling of the number of particle–particle force calculations, but this avoids the penalty of an order of magnitude involved in reductions. Indeed the following version of the algorithm runs much faster than algorithm (a):

algorithm (b1): direct summation for the Connection Machine (1)

```
do j = 1,N
    do i = 1,N in parallel
        if (i.ne.j) then
            do k = 1,3
                dx(i,k) = x(i,k) - x(j,k)
            enddo
            r2(i) = dx(i,1) * *2 + dx(i,2) * *2 + dx(i,3) * *2
            r3inv(i) = 1/(r2(i) * sqrt(r2(i))
            do k = 1,3
                a(i,k) = a(i,k) - dx(i,k) * (m(j) * r3inv(i))
            enddo
        endif
    enddo
enddo
```

Here, the fact that the loop over i is executed in parallel means that processor i holds in its local memory the values of $x(i,1)$, $x(i,2)$, etc, and operates on them. With this algorithm, in the jth iteration all processors require data from processor j. This broadcasting is performed by the host machine. The host machine first reads the memory of the jth processor and then broadcasts its value to all processors. This broadcasting is very fast. Thus this algorithm can extract nearly the full performance of the Connection Machine. Figure 3b shows how this algorithm works. The meaning of the lines are the same as in fig. 3a.

We can also make use of the symmetry in the gravitational force through the following algorithm:

algorithm (b2): direct summation for the Connection Machine (2)

```
do l = 0,N/2 - 1
    do  = 1,N in parallel
        j = mod(i + l, N) + 1
        do k = 1,3
            dx(i,k) = x(i,k) - x(j,k)
        enddo
        r2(i) = dx(i,1) * *2 + dx(i,2) * *2 + dx(i,3) * *2
        r3inv(i) = 1/(r2(i) * sqrt(r2(i))
        do k = 1,3
            a(i,k) = a(i,k) + dx(i,k) * (m(j) * r3inv(i))
        enddo
        if(2*l.ne.N - 2) then
            do k = 1,3
                a(j,k) = a(j,k) - dx(i,k)*(m(j)*r3inv(i))
            enddo
        endif
    enddo
enddo
```

Figure 3c illustrates this algorithm. In each step of the iteration, each processor requests data from one other processor (in such a way that no two processors request data from the same other processor). Then each processor calculates the displacement vector and distance, accumulates the force, and sends back the distance and the displacement vector to the processor from which it got the position and mass. Then each processor again accumulates the force (which is symmetric with respect to the previous force accumulation), using the data it just received. The amount of calculation is reduced from algorithm (b1) by a factor of nearly two. However, the time to execute this algorithm on the Connection Machine is actually longer than that of algorithm (b1). The reason is that this algorithm requires interprocessor communication, which is considerably slower than broadcasting via the host machine.

We now introduce another algorithm which also requires interprocessor communication, but with a simple fixed pattern. With this algorithm, the pattern of communication is a one-dimensional ring. The positions and masses are passed around the ring.

algorithm (b3): ring algorithm for direct summation

```
do i = i,N in parallel
    work_mass(i) = mass(i)
    next_in_ring(i) = mod(i,N) + 1
    do = 1,3
        work_pos(i,k) = x(i,k)
    enddo
enddo
do 1 = 0,N - 1
    do i = 1,N in parallel
        work_mass(next_in_ring(i)) = work_mass(i)
        do k = 1,3
            work_pos(next_in_ring(i),k) = work_pos(i,k)
        enddo
        do k = 1,3
            dx(i,k) = x(i,k) - work_pos(i,k)
        enddo
        r2(i) = dx(i,1)**2 + dx(i,2)**2 + dx(i,3)**2
        r3inv(i) = 1/(r2(i)*sqrt(r2(i)))
        do k = 1,3
            a(i,k) = a(i,k) - dx(i,k)*(work_mass(i)*r3inv(i))
        enddo
    enddo
enddo
```

Table 4
Comparison of direct summation algorithms on CM

Algorithms	Total time	Calculation	Communication
broadcast (b1)	15.9 s	12.6 s (79%)	3.3 s (21%)
$N(N-1)/2$ algorithm (b2)	29.9 s	8.6 s (29%)	21.3 s (71%)
ring algorithm (b3)	19.9 s	12.8 s (64%)	7.1 s (36%)

Note: The time is the elapsed time per one timestep. The time for communication is measured by commenting out all statements involving the calculation.

Fig. 3d illustrates this algorithm. The pattern of the calculations is similar to that of algorithm (b2), except that the present algorithm does not make use of the symmetry of the force. Here, the amount of communication required is exactly the same as with algorithm (b2). Nevertheless, with the actual Connection Machine, the time consumed for communication is significantly smaller. The reason is that with the ring pattern, most processors communicate with physically near processors within the network.

Table 4 gives the time required for communication and calculation for each algorithm described above for the CM-2 with a 4 MHz clock speed (the speed of the machine when we did our benchmarking), with 8k processors with floating-point accelerators. The number of particles used is also 8k, to obtain the peak performance of the machine.

From table 4 we can see that algorithm (b1) is the best and (b2) is the worst. Algorithm (b1) is fastest since the communication time needed is smallest. However, the difference is relatively small, less than a factor of two. This means that we should consider other factors to determine which algorithm we should use. With algorithm (b2) and (b3), all communication is performed by the processor array itself. The host machine simply issue the commands. On the other hand, the host fetches and broadcasts data with algorithm (b1). Thus, with algorithms (b2) and (b3) we can run several simultaneously, if the number of particles N is smaller than the size of the machine. Moreover, we can make several copies of one N-body system. Then, each copy of the system can calculate a small part of the interaction. After each copy is finished calculating its own contribution, all results are added to the original N-body system. If the number of copies is much smaller than N, the overhead of making copies and summing the force is negligible.

3.2.3. *Performance comparison*

Fig. 4 gives the performance of the Connection Machine and several other supercomputers for the direct summation algorithm. For each machine we express the speed in units of Mfocs, where 1 Mfocs stands for one million force calculations per second. Here one force calculation is defined as the computation of one interparticle acceleration, without making use of the symmetry in the acceleration felt by each particle separately. Thus one time step for a N-body system involves $N(N-1)$ force calculations in our definition. For those algorithms which actually use the intrinsic symmetry, and calculate only half as many accelerations, we simply double the measured rate of force calculations to arrive at a Mfocs number. This procedure is fair from the point of view of a user, who is not interested in implementation details of the algorithm, but instead wants to know the number of time steps which can be performed per second. In our benchmarks, we prefer to give a Mfocs number, rather than the direct number of

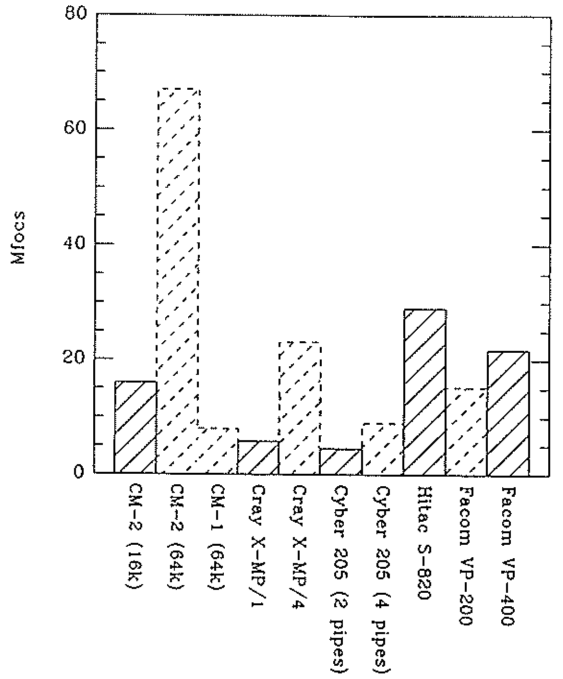

Fig. 4. Performance of the direct summation algorithm on various computers. Unit is Mfocs (million force calculations per second). Solid bars are obtained by measurement. Dashed bars are estimated.

timesteps, since the latter strongly depends on the total particle number N, while the former is only weakly dependent on particle number.

In fig. 4 we show the speed as measured from a 8192-body system. Algorithm (a) was used for all supercomputers. The speed is in million force calculations per second (Mfocs). It is defined as:

$$R_{\text{force}} = \frac{N(N-1)}{T_{\text{step}}},$$

where T_{step} is the CPU time in seconds per time step. For the Connection Machine we show the measured speed for the optimized version of Algorithm (b3) and the estimated time for a system with 64k processors (which is the maximum configuration currently available). We can see that even one-quarter of the full Connection Machine offers a performance comparable to that of present-day supercomputers.

In table 5 we list the results of several experiments which show the performance for each machine. The number of particles used here is 8k. The first column shows the number of forces calculated per second in unit of Mfocs (million force calculations per second). The extrapolated performance for the full configuration of 64k processors is impressive. It outperforms all vector processors in this table.

The second column gives the effective floating point speed F_{real} in real Mflops. To obtain the numbers in this column, we have counted the number of floating-point operations. The number

Table 5
Comparison of the performance for direct summation

Machine	R_{force}	F_{real}	F_{cor}	F_{max}	e_{sup}	e_{grav}	e_{prac}
CM (8k, 4 MHz)	4.2	118	139	625	0.19	3.6×10^{-2}	6.7×10^{-3}
CM (64k, 8 MHz)	67	1883	2228	10000	0.19	3.6×10^{-2}	6.7×10^{-3}
Cray X-MP/18	5.8	106	121	210	0.50	5.5×10^{-2}	2.8×10^{-2}
Cyber 205 (2 pipes)	4.5	88	94	200	0.44	5.1×10^{-2}	2.3×10^{-2}
Hitac S-820/80	29	889	618	3000	0.32	3.3×10^{-2}	9.7×10^{-3}
Facom VP-400	22	633	375	1160	0.55	3.5×10^{-2}	1.9×10^{-2}

of division and square root operations is multiplied by weighting factors which reflect the performance of each machine for these particular operations. The values used are listed in table 6. In the third column we present the corrected Mflops F_{cor}, in which the weights for division and square root are scaled to be the same for all machines. The assumed values are 5 for the division and 10 for the square root. This gives a more reasonable basis for comparison than real Mflops, though it depends on the assumed values for the factors. In the next column we also give the peak speed in Mflops F_{max} as advertised by the manufacturers.

One measure of the efficiency of a code on a machine is given by the ratio between the actual Mflops obtained by the code and the peak Mflops rating of the machine. Denoting this superficial efficiency measure by e_{sup}, we have

$$e_{\text{sup}} = \frac{F_{\text{real}}}{F_{\text{max}}},$$

which is tabulated in table 5. This value, however, does not reflect the actual speed at which the specified problem can be solved. One reason is that the algorithm used on the Connection Machine is somewhat different: it requires more computation to obtain the same result. Therefore a mere comparison of measurements in terms of Mflops is not sufficient. Another reason is that the ratio of the speed of the elemental operations differs from machine to machine, as we can see from the difference in F_{real} and F_{cor}. The net effect of these factors can be expressed by introducing a gravitational efficiency measure as the inverse of the number of

Table 6
Factors used to calculate F_{cor}

Machine	Divide	Square root
CM	3.6	6.3
Cyber	6.25	6.25
Cray	2.73	5.32
Hitac	3.95	29.5
Facom	2.93	40.9

floating-point calculations needed to compute one particle–particle force:

$$e_{\text{grav}} = \frac{R_{\text{force}}}{F_{\text{real}}}.$$

Finally, the practical efficiency of a computer, expressed as the number of practical results obtained per unit of peak speed, is given by

$$e_{\text{prac}} = e_{\text{grav}} e_{\text{sup}} = \frac{R_{\text{force}}}{F_{\text{max}}},$$

and tabulated in table 5. Note that the Connection Machine yields a practical efficiency comparable to other supercomputers.

We conclude that the performance of the Connection Machine CM-2 exceeds that of current vector supercomputers for a direct summation approach to N-body calculations, for N large enough. As mentioned in section 3.2.2, even for values of N smaller than the number of available processors, careful coding can reach a comparably high performance. For large N, the most appropriate algorithm for the Connection Machine is actually simpler than that for vector or scalar processors.

For completeness, a few technical remarks are in order: (1) On some vector processors, the cost of the square root operation is rather high, and forms a bottleneck in the force calculations. In appendix A we discuss a way of speeding up the square root operation, using a Newton–Raphson approach. When we implemented this method on a Hitachi and a Fujitsu, we found a speed up of the force calculation of 30 ~ 40%. (2) Our benchmarks on the Connection Machine stem from a carefully optimized code, written mostly in low-level language (the assembly language PARIS, rather than C*), since the C* compiler did not yet produce well optimized code. The difference in speed between the hand-optimized PARIS-C* version and the pure-C* version is about a factor of two.

What is the implication of our benchmarks for astrophysical applications? We have found a reasonably high efficiency of the relatively simple algorithms incorporating direct summation, both on supercomputers and on the Connection Machine. This is encouraging when tackling problems with a relatively small number of particles, $N \approx 10^3$ or even up to $N \approx 10^4$ on the fastest machines. However, for many astrophysical problems a larger number of particles is needed. For example, we need $10^5 \sim 10^6$ particles to model a collision of two galaxies in detail. Even with the full 64k configuration of the Connection Machine, one time step for such a simulation would take 0.1 ~ 10 hours. A typical simulation would require at least 10^4 time steps. This implies a total run time of 0.1 ~ 10 years. To make matters even worse, a detailed modeling of this kind requires a large number of experiments, say a hundred or so, to study the dependence of the results on the various input parameters. This leaves us with an estimate of 10 to 1000 years, a clearly unpractical situation. Fortunately, a tree code can achieve much better performance here, even with a much lower nominal efficiency in terms of Mflops, simply because the number of force calculations required can be reduced by orders of magnitude, since the computational complexity is $O(N \log N)$ instead of $O(N^2)$ for direct summation codes.

3.3. *Hierarchical tree algorithm: an irregular communication pattern*

3.3.1. *The tree code*

The discription of the tree algorithm can be found in ref. [12]. The basic idea is to surround an individual particle with groups of distant particles, and to compute only one force term which approximates the force from whole group acting on that individual particle. The bookkeeping for allocating groups is performed by dividing space into an oct-tree. The whole system is placed within a master cube, which is divided into eight subcubes. This dividing process is continued locally and recursively until at most one particle resides in each (locally) smallest subcube. These subcubes containing individual particles form the leaves of the tree. For each individual particle, groups are chosen as nodes in the tree structure, such that the ratio of their size and distance does not exceed a prescribed criterion, in a way which can be analyzed regorously. The force from such a group of particles is then determined by putting the total mass of the group in the center of mass of that group, thereby neglecting quadrupole and higher multipole contributions of the force from that group. The magnitude of these multiple errors is bounded by the prescribed criterion. Details are discussed by Barnes and Hut [12,15] and Hernquist [16]. Below we will indicate a group which obeys this separation criterion with respect to a particle as being well separated from that particle. The force calculation part of the tree algorithm then reads:

algorithm (c1): recursive tree-force calculation for particle *i*

```
subroutine treeforce(i)
    node = root_node
    if (the node and particle i are well separated)
        force = force from the total mass in the center of mass of the node
    else
        force = sum of forces from the children of the node
    endif
    return
```

We use the following condition to determine if a node is well separated from a particle:

$$\frac{l}{d} < \theta,$$

where l is the size of the cube corresponding to the node, d is the distance between the cube and the particle, and θ is the parameter which determines the accuracy and thereby the amount of computation needed. Fig. 5a shows how this algorithm works. To obtain the gravitational force on the particle indicated by X, we start from the root of the tree. The root is usually not well separated from the particle. Therefore, we descend the tree and try to evaluate the total force as a sum over the forces from the children of the root. If a child-node is well separated, its contribution to the force is evaluated. If not, we further descend the tree recursively until we reach nodes which are well separated or leaves which contain single particles (which are by definition well separated).

Although the above algorithm is simple and straightforward, it is not easy to implement on vector processors or on the Connection Machine. One of the problems with vector processors is

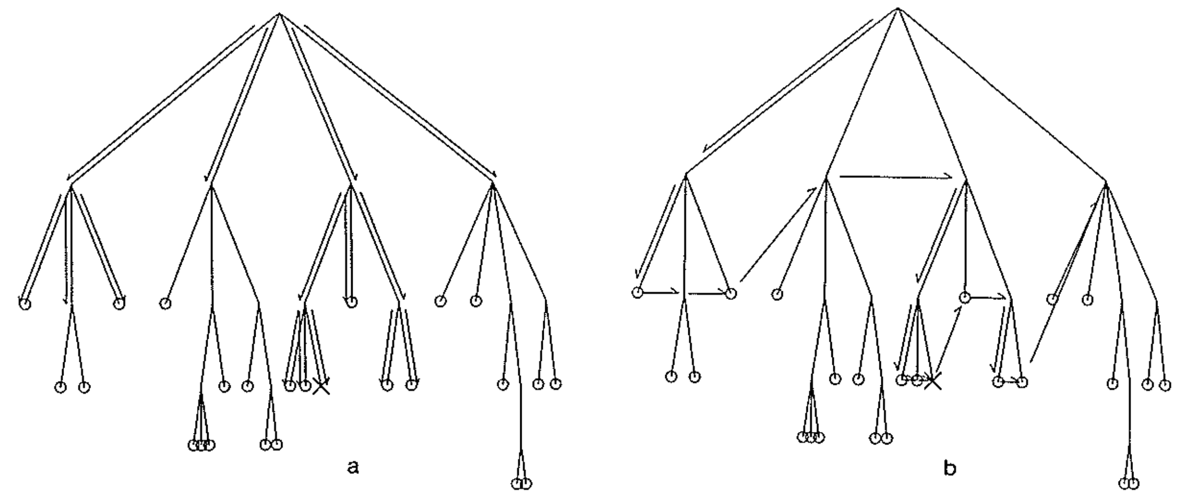

Fig. 5. Force calculation using the tree. Small circles indicate particles. X indicate the particle on which the force is calculated. Arrows indicate how the calculation proceeds. (a) is for the recursive algorithm (c1) and (b) is for the iterative algorithm (c2).

that the only language which runs efficiently is Fortran, which does not allow recursive functions. Another problem is that other languages which allow recursion (including some existing Fortran dialects, as well as the expected new standard Fortran 8X) are not likely to produce efficient code for algorithm (c1), because recursive function calls are difficult to vectorize. Thus we have to modify (c1) significantly, in order to use it on vector machines.

On the Connection Machine, a language such as C* does allow recursion. However, a straightforward implementation of algorithm (c1) would show a very low efficiency. Note that on the Connection Machine, each particle is assigned to an individual (virtual) processor. In this way, the forces on all particles will be calculated in parallel, since each processor executes algorithm (c1) simultaneously. At first, starting at the root, all processors will be active. However, at each level of descent in the tree, when a certain cell is further subdivided, some processors will become idle. This happens to those processors which are associated with particles for whom that particular cell does not need to be subdivided further. By the time we reach the smallest cells, the large majority of the processors are idle, and only a few processors, representing some of the near neighbors to that cell, are still active. But when we realize that the large majority of the cells actually are small and only contain a few particles, then we see immediately how terribly inefficient a straightforward implementation of algorithm (c1) is. To be specific, mosts cells have one or more neighbor particles that require subdivision of that cell. The number of cells is O(N). Thus the computation time would also be O(N), which would completely nullify the advantage of the O(log N) algorithm, by degrading it down to a parallel N^2 algorithm. Therefore, we are forced to modify the simple algorithm (c1) considerably for the Connection Machine as well.

The alternative for the recursive implementation of the tree walk is to switch to some form of iterative procedure. The most obvious choice is the explicit construction of a stack, which simulates the recursive calls implied in the above algorithm. Another possibility, which avoids the use of a stack altogether, is the algorithm described by Makino [17]. These two different approaches to an iterative tree walk can be characterized by the different ways of finding one's way out of a maze. The first approach corresponds to keeping a record of each previous choice at each intersection, and to work one's way through all choices (keeping an explicit stack). The second approach would be to follow systematically the wall at the right-hand side, say, which would lead to a complete traversing of the maze as well (without the need for explicit bookkeeping). Of course, a tree walk only visits O(log N) nodes, instead of all O(N) nodes; but otherwise the analogy holds. Fig. 5b illustrates the second approach in the case of a tree walk.

The structure of the stack-less iterative force calculation is as follows (see ref. [17]):

algorithm (c2): force calculation through a stack-less iterative tree walk

```
subroutine treeforce(i)
    node = root_node
    while (node .ne. null)
        if (the node and the particle are well separated)
            force = force + (force from the center of mass of the node)
            node = next(node)
        else
            node = first_child(node)
        endif
    endwhile
    return
```

Here we assume that the children of a node form an ordered set. *first_child* returns the first element of the set of children of a node. The function *next* is a bit more complicated: it can return a brother-node, or an uncle-node, or a borther-of-grandfather-node, etc., depending on where the linear mapping of the tree continues. To be precise, next is defined as follows:

```
function next(node)
    if (the node is not the last member of the children list of its parent)
        next = the next member of the parent's children list
    else
        next = next(the parent of node)
    endif
    return
```

Before discussing the implementation of the stack-less algorithm on vector processors and on the Connection machine, let us make a short digression to explain our preference of a stack-less algorithm over the more natural stack-based iterative scheme. The reasons are twofold: a greater simplicity of the program, and a significant reduction of the communication overhead resulting in a speed-up of a few tens of percent on vector machines, and of a factor two on the Connection Machine.

The simplicity of the stack-less code is easily seen. The only bookkeeping is done through a mapping of the tree, immediately after its construction, in the form of a uniquely determined linear unfolding. This is much simpler than the bookkeeping which has to be performed in the case where an active stack has to be kept which changes dynamically throughout the tree walk. This dynamic bookkeeping involves pushing and popping the stack, operations which require indirect addressing and a very careful coding to achieve efficiency under vectorization.

The speedup gained from a stack-less code, compared to a stack-based iteration, is a subtle effect which hinges on the use of a SIMD machine to simulate a MIMD computer. In the stack-based code, whenever we inspect our new node for each particle in parallel, some particles will want to open that node to descend to the children, while other particles will want to skip the children. On a parallel SIMD machine, however, all processors must obey the same instruction, or else be idle. In either case, each particle incurs a penalty in computing time equivalent to requesting the information needed to evaluate both the if and the else side of a conditional statement. Therefore, the effective amount of computer time needed corresponds to a situation where each processor requests the addresses of all eight children of the node it is currently looking at, irrespective of whether it will want to use that information. Let us now compare this situation with that of the stack-less code. Upon inspecting a new node for each particle in parallel, again each node receives the full information associated with the next links in the chain. The important difference with the stack-based code is the number of links: instead of eight children, we now receive the information of only two links: the first child and the next node. Thus we gain a factor of four in speed for this part of the communication. On the Connection Machine, it turns out that for the stack-based version roughly half the time required for communication is actually spent in requesting the addresses of new links to be searched; therefore, switching to a stack-less code gained us a factor of about two in speed (since the speed of the Connection Machine CM-2 in the case of tree codes is heavily communication-bound; see below).

3.3.2. Vector processors

On would expect no difficulty in principle in vectorizing the algorithm (c2), because the force calculations are executed completely independently for each particle. However, in practice it is necessary to modify the algorithm significantly to obtain good performance. The main difficulty stems from the lack of descriptive power of Fortran 77 and from the inability of vectorizing compilers to recognize when complex code for scalar machines is vectorizable. Another compilation lies in the fact that individual machine characteristics require special modifications to the basic algorithm to extract optimal speed. Let us discuss these points in some detail.

The force calculation part of a tree code for scalar processor has the following form:

algorithm (d1): the basic form for a hierarchical force calculation

```
do i = 1,n
   call treeforce(i)
enddo
```

Here the subroutine treeforce stands for the algorithm (c2). Most vectorizing compilers cannot deal with a DO loop containing subroutine calls. Thus, we are forced to expand the subroutine inline:

algorithm (d2): subroutine call expanded

```
do i = 1,n in parallel
    while (some condition)
        do something
    endwhile
enddo
```

Due to syntax limitations in standard Fortran 77, the while ... endwhile structure needs to be implemented as a combination of an IF statement and a GOTO statement. The next stumbling block is the fact that vectorizing compilers cannot vectorize DO loops containing a backward GOTO. Nevertheless, we can vectorize this by using yet another trick:

algorithm (d3): while moved out of the loop

```
while (some condition is true for any of index i)
    do i = 1,n in parallel
        if (the condition)
            do something
        endif
    enddo
endwhile
```

Note that the first line can be implemented by a vector-if statement and an or-reduction over the vector. This type of structure is vectorizable by all Japanese compilers, but not by most American compilers. We should remark here that the efficiency of this algorithm has degraded for use on sequential machine, because of the extra operations added in order to make the algorithm vectorizable. Thus, we cannot use the same program on scalar and vector machines, for this algorithm.

Moreover, there still remains room for improvement. The treatment for the IF statement varies for different compilers. For the simplest case, all statements under control of the IF statement are compiled using masking. Only the vector elements with a true flag are subsequently processed. This form of execution is similar to conditional execution on the Connection Machine. In this simplest case, the masked operation requires an amount of time proportional to the vector length, independent of the amount of calculation actually performed. Thus, if the fraction of all vector elements which need to be processed is small, the performance degrades significantly, as was discussed in section 2.2. If the compiler is smart enough, it can separate the decision process from the state at which actual calculations are performed:

algorithm (d4): vectorization of IF using a list vector

```
nlist = 0
do i = 1,n in parallel
    if (some condition)
        nlist = nlist + 1
        list(nlist) = i
    endif
enddo
do i = 1,nlist in parallel
    do something for list(i)
enddo
```

The first loop produces a list of indices which indicates where further work is needed. The second loop execute the actual work. With this structure, the O(N) part of the DO loop is limited to the first loop. In the second loop, the number of iterations is smaller than the length of the original vector. For those compilers which do not automatically translate (d3) to (d4), we can improve the efficiency by explicitly rewriting the DO loop containing an IF statement in the form of algorithm (d4). Of course, this will result in a further degrading of the performance of that code on a sequential machine.

Finally, some compilers cannot vectorize any DO loop which contains an IF statment. For these machines, one should rewrite one's program completely by using the "vector extensions" provided by the manufacturer. Unfortunately, at present, no standard exists for the syntax of "vector extensions". This implies that code written for a specific machine will run on neither scalar machines nor other vector machines.

3.3.3. The Connection Machine

Parallelization of the algorithm (c2) on the Connection Machine is quite straightforward:

```
do i = 1,n in parallel
   call treeforce(i)
enddo
```

With these instructions, the compiler automatically produces codes corresponding to algorithm (d2). In this straightforward implementation, each particle is mapped to one processor. Each node of the tree is also mapped to one processor. Therefore, the requirement for the number of processors is larger than the number of particles N. In the force-calculation routine, only processors mapped to the particles are active, while processors mapped to the nodes of the tree remain idle. Thus, the efficiency of this algorithm is limited by the ratio between the number of processors attached to the particles and the total number of processors available. This ratio is typically about 50%.

Unfortunately, a straightforward implementation of the above algorithm turned out to be extremely inefficient due to software limitations on the Connection Machine at the time of our benchmarking. The difficulty appears whenever many processors simultaneously request the data in the same processor. In each iteration of the body of the while loop in algorithm (c2), every active processor tries to read the memory of some other processor. In the initial step, all processors assigned to particles request the data of the root node, thereby trying to access the same procsssor. In general, nodes in the higher levels of the tree are accessed by larger number of particles than the nodes in the lower levels. Therefore, it is inevitable that the nodes in the higher levels are accessed by many particles simultaneously. With the current software implementation for the interprocessor communication on the Connection Machine, all requests for a single processor are processed one at a time. Thus it takes an amount of time which scales as O(N) in the worst case in which all processors request the data in one processor. Software improvements are under development which should reduce this amount of time to O(log N). The maximum speed possible, however, will not exceed the speed of communication without collision, i.e. the speed of communication when each processor request the data from a different processor.

Faced with these current softwares, we first give a theoretical estimate of the performance which may be obtained with new communication software. Then we discuss the speed obtained in our actual implementation which contained stopgaps to work around the current software limitations.

The number of words which one particle requires for one iteration in algorithm (c2) is seven. It needs the position (3 words), the mass, the size of the cell, and the pointers to the next node and the first child. The length of the message which can be send in one operation is currently limited to 6 single precision words (192 bits). With a little bit of hand-coding we can pack these seven pieces of data into six words. Therefore we need only one GET operation for one iteration in algorithm (c2). The time required to execute one GET operation is 6 ms for a machine running on a 4 MHz clock. Note that this implies a communication speed of 1000 words/s, which is considerably larger than the 300 words/s implied from table 3 for the equivalent operation a(i) = b(j(i)). The reason is that the time needed to execute one GET operation for a message with a length of 192 bits is significantly shorter than that required to execute six GET operations each for a message with a length of 32 bits.

The number of the floating-point operation is around 30 per iteration in algorithm (c2). In other words, the amount of time needed for actual computation is 1.5 ms per iteration in algorithm (c2) for a 4 MHz clock. Therefore, the maximum number of force calculations which can be evaluated by a 64k-processor Connection Machine with a 8 MHz clock is $8/4 \times 65{,}536/(0.006 + 0.0015) = 1.7 \times 10^7$ or 540 Mflops. This is the maximum peak performance possible with present hardware. To be more precise, we should take into account two factors: (a) not all processors are attached to particles, (b) some processors finish their work before others. Typically both cause a loss of efficiency of about 50%. Thus, we predict a optimal performance of the tree code on the Connection Machine of about 140 Mflops, when the current software bottlenecks are removed.

One possibility for working around current software limitations on the Connection Machine is to maintain many copies of those nodes which are accessed frequently. For example, if each processor keeps the information of the root node, we can avoid collisions at the initial step altogether, because each processor can find the information needed within its local memory. We have implemented this idea as follows. After constructing the tree, we count the number of nodes for each level of the tree. The number of copes $N_{\text{copies},l}$ desired of each node at level l in the tree was simply taken to be

$$N_{\text{copies},l} = \left[\frac{N}{N_{\text{nodes},l}} \right],$$

where N is the total number of particles, and $N_{\text{nodes},l}$ is the number of nodes for level l of the tree.

With 8k particles, the worst case of processor access conflict for a typical iteration of algorithm (c2) occurs with about ten processors requesting data from a single processor. With this multiple-copy method, we reached a speed of about 25 iterations of algorithm (c2) per second, for a Plummer model, with a clockspeed of 4 MHz. This implies an effective speed which is slower than the theoretical limit by a factor of 6. However, with our multiple-copy method, all processors can be attached to particles. Thus we have obtained an relative improvement of efficiency of a factor two. Therefore, our modified code is slower than the theoretical limit by only a factor of 3, which is quite satisfactory for a temporary fix. The number of iterations of algorithm (c2) per force calculation is about 500 for 8k particles. Thus, a force calculation takes about 20 seconds.

The well optimized direct summation code which we discussed in section 3.2.2 takes only 16 seconds per time step and is still somewhat faster than the tree calculation. For the larger N, the tree algorithm becomes more efficient, because on the Connection Machine the scaling of the computation time is $O(\log N)$ for a tree method and $O(N)$ for the direct summation method, as long as the number of particles remains less than the number of processors. The cross-over point between the tree method and direct summation thus occurred around $N \approx 10^4$ at the time we used the Connection Machine.

3.3.4. Performance comparison

The tree algorithm offers a severe test of the flexibility of both hardware and software of fast computers, because it involves a very complex type of memory access. Since the pattern of tree interconnections exhibits a complex structure which is rapidly changing, no fixed patterns of contiguous memory storage can be assigned to the variables used in the force calculations. To make things worse, the higher levels of the tree form bottlenecks for memory access.

Given these intrinsic problems, it is not surprising that all fast computers suffer a significant loss of speed, when we switch from the simple direct summation algorithm to the tree algorithm. Just how serious this performance degradation is, can be seen from fig. 6, where the highest Mfocs rate attained for a tree code is lower than that attained for direct summation codes by a

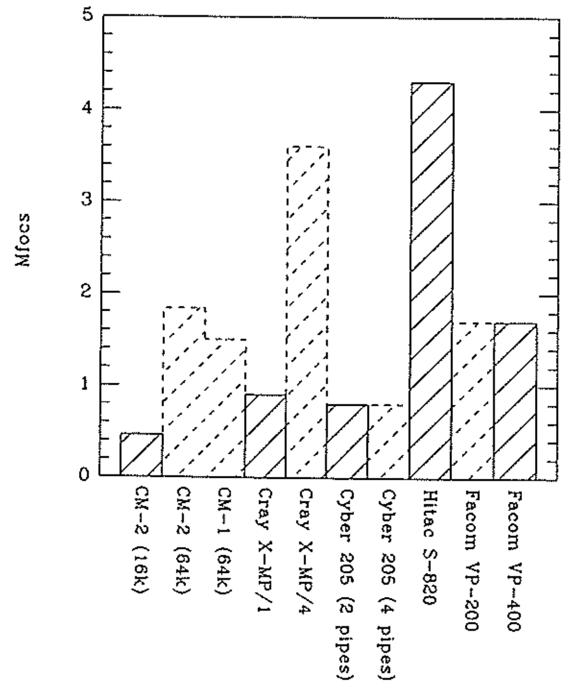

Fig. 6. Performance of the tree algorithm on various computers. Unit is Mfocs (million force calculations per second). Solid bars are obtained by measurement. Dashed bars are estimated.

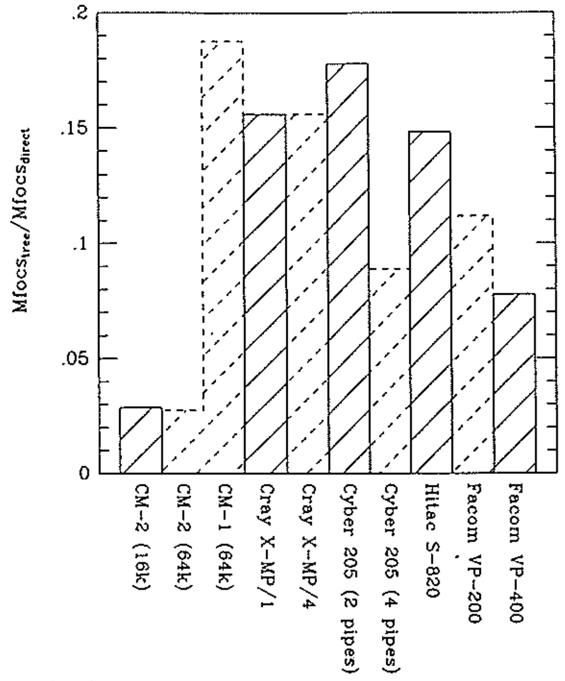

Fig. 7. Relative speed of the tree algorithm compared to that of the direct summation algorithm on various computers. Solid bars are obtained by measurement. Dashed bars are estimated.

factor of more than 15. Interestingly, in each category of algorithms, the highest speed is attained by a completely different type of machine. When we inspect individual machines, we see a highly machine-dependent performance degradation, as indicated in fig. 7.

We can see that the performance of the Connection Machine with 8k processors is significantly lower than that of vector processors. The estimated speed for a full 64k configuration is comparable to that of vector processors. Table 7 gives more detailed information about the performance. The entries are of the same type as those given in table 5. We can see that both F_{real} and F_{cor} for the tree algorithm are much smaller than those for the direct summation algorithm. Most vector machines, as well as the 64k Connection Machine, show a similar speed.

Table 7
Comparison of the performance for treecode

Machine	R_{force}	F_{real}	F_{cor}	F_{max}	e_{sup}	e_{grav}	e_{prac}
CM (8k, 4 MHz)	0.12	3.8	4.4	625	0.006	3.1×10^{-2}	1.8×10^{-4}
CM (64k, 8 MHz)	1.84	60	70	10000	0.006	3.1×10^{-2}	1.8×10^{-4}
Cray X-MP/18	0.9	26	31	210	0.12	3.5×10^{-2}	4.3×10^{-3}
Cyber 205 (2 pipes)	0.8	25	30	200	0.12	3.2×10^{-2}	4.0×10^{-3}
Hitac S-820/80	4.3	225	146	3000	0.075	1.9×10^{-2}	1.43×10^{-3}
Facom VP-400	1.7	112	60	1160	0.096	1.52×10^{-2}	1.5×10^{-3}

In terms of efficiency, the Connection Machine comes out worst. This is partly because of current software limitations, as discussed in section 3.3.3. The new software for the interprocessor communication may speed up the tree algorithm on the Connection Machine by factor of about 3. Even with that speed, however, the efficiency of the Connection Machine is still significantly lower than that of the vector machines.

Thus the difference in architecture is responsible for the difference in efficiency. For most vector machines discussed here, the bottleneck is the indirect addressing. On the Connection Machine, too, the same part of the algorithm leads to a bottleneck, namely in the interprocessor communication. As discussed in section 2, the force calculation itself is much faster on the Connection Machine than on the vector processors. Nevertheless, the indirect addressing in the vector processors and the interprocessor communication in the Connection Machine show a similar speed. This explains why the Connection Machine exhibits a rather low efficiency compared with vector processors.

Note that even if communication would require no time at all, the efficiency of the Connection Machine could still not be 100%, unlike vector processors. The reason for the inefficiency of the Connection Machine lies in the fact that it is very difficult to keep all processors active, as discussed in section 3.3.3. With vector processors, we can reduce the loss in efficiency caused by an IF statement by using a list vector (see section 3.3.2). On the Connection Machine, an IF statement causes a fraction of the processor to remain idle, and there is no general way to make up for this loss in efficiency. This does not imply that the Connection Machine is less suitable for a complex algorithm than the vector processors. The most relevant characteristic of a machine is the actual speed at which calculations are carried out, and not the efficiency relative to the peak performance of that machine. Any specific limitation in the efficiency of the architecture remains unimportant, as long as the actual performance is high enough. In this sense, the performance of the Connection Machine is perfectly acceptable, since it is already faster than presently available supercomputers.

The next generation of supercomputers, such as the Cray-3 and the ETA-10 may or may not outperform the CM-2. However, the Connection Machine will also evolve. Moreover, there seems to be more room for improvement for the Connection Machine because it is a completely new type of computer. Most importantly, increasing the number of processors in the Connection Machine will not require any change in existing codes, whereas further parallelization of supercomputers introduces new complications which are still largely unexplored.

4. Software

In section 3 we investigated the performance of different computers in terms of speed of execution of two different types of N-body algorithms. Although such a performance comparison forms an important part of the evaluation of different machines, there are other factors which play an important role in establishing the overall usefulness of a computer. On the hardware level, the reliability is an important issue. On the software level, there is a vast range in flexibility and expressiveness of the different languages and dialects offered on current supercomputers and parallel computers. Good debugging tools, which are essential for writing and maintaining complex codes, form another important ingredient in the efficiency with which a computer can be used. Also, the frequency and character of bugs in the operating system and in the compilers contributes significantly to the overall usefullness of a computer. Finally, stability of the language definition of a vector/parallel extension of an existing language is crucial to guarantee future use of complex codes.

A comprehensive discussion of quality and reliability issues of both hardware and software lies beyond the scope of the present paper. Concentrating solely on speed of execution of user programs, however, would be too limited from the point of view of a scientist trying to select the most useful computer on which to carry out his calculations. Therefore, we have decided to report here some personal reflections concerning the programming environments on vector processors and on the Connection Machine.

4.1. Languages

The two most important considerations in choosing a computer language are flexibility and historical continuity. Flexibility in adapting an algorithm in an optimal fashion to the hardware of a machine requires a language with special vector/parallel syntax constructions. Flexibility in allowing the programmer to express an algorithm in a natural, abstract and modular fashion requires a modern language based on data abstraction, such as C and Pascal. Historical continuity, however, is equally important and is often diametrically opposed to the two requirements of flexibility mentioned before.

There is basically no good solution to the opposing requirements of continuity and flexibility, and different problems require a different emphasis. For example, it is of crucial importance to be able to run on a new machine an existing fluid dynamics package written in standard Fortran with an investment of several person-years. As an altogether different example, consider a researcher who is starting to implement from scratch a novel algorithm on a new machine. In the latter case, concern will shift to flexibility requirements, and a parallel C version may be the language of preference.

Currently, three types of languages are used on vector processors and on the Connection Machine: (1) standard Fortran 77 with automatic vectorization; (2) Fortran 77 with special vector extensions; (3) language other than Fortran with vector/parallel extensions. Compilers in the first category have been supported from the beginning on all Japanese supercomputers, but only recently on some American vector machines as well. Compilers for Cray and CDC supercomputers fall mainly in the second category. Compilers for the Connection machine currently fall in the third category, providing vector extensions for Lisp and C. In addition, C compilers are beginning to become available on several supercomputers as well. In the following subsections, we discuss each language category separately.

4.1.1. Standard Fortran

Using standard Fortran on a supercomputer poses a severe task for a compiler, which should be able to find the potential parallelism in a sequential Fortran program in order to apply vector operations wherever possible. The most profitable targets for vectorization are those pieces of code where a single block of instructions is repeatedly executed. In Fortran such looping constructs are usually expressed through DO loops. Therefore, standard Fortran compilers generally take as their targets for vectorization only DO loops. As a consequence, this process of automatic vectorization is usually called DO loop vectorization.

Automatic vectorization is a very useful tool since it guarantees continued use of previously written Fortran programs. In principle, automatic vectorization could further improve programmer productivity also when writing new codes, since it attempts to make hardware efficiency considerations transparent to the user. One of the most important advantages of such transparancy is the possibility to develop and debug codes on a scalar computer, such as a workstation, before running it on a supercomputer.

In practice, however, automatic vectorization often causes a severe loss of efficiency for any but the simplest codes. For complex codes, such as the tree codes tested in section 3.3, automatic vectorization reduced the speed of any supercomputer which we tested down to the lowest possible speed, namely the scalar speed of the processor. But even for relatively simple and straightforward algorithms, such as are used in the direct summation codes, special care has to be taken to guide the compilers into recognizing vectorizable DO loops. In section 3.2 we mentioned that a simple direct summation algorithm, written for a workstation or a mainframe, would give terrible performance on any vector machine with automatic vectorization, unless a number of changes in the algorithm were put in by hand, leading to algorithm (a).

For complex algorithms, the modifications to a scalar code are more extensive, in some cases leading to a complete revisions of the control structure, as discussed in section 3.3. For both types of algorithms, we saw how the requirement that DO loop vectorization should be recognizable by the compiler led to unnatural coding. In particular, the fact that subroutines are not allowed in DO loops which are targets for automatic vectorization makes it very difficult to write a clean code.

Our conclusion is that current vectorizing compilers have not yet reached their goal of being able to vectorize relatively complicated algorithms written in a natural way, with clarity for the human reader taking first priority. Although vectorizing compilers minimize the extent to which a scalar code has to be rewritten, the necessary modifications are substantial enough to make the resulting vectorizable code unusable on the original scalar machines. The advantages of vectorizing compilers are still substantial for typical cases in which we have a large code in which most of the computations are done in a relatively small part of the program, so that we only have to modify the compute-intensive part.

4.1.2. Vector extensions for Fortran 77

When we want to exert more direct control over the vectorization process, it is possible on most vector machines to use a version of extended Fortran which includes special vector instructions. These language extensions have two main advantages and two main drawbacks. The major advantage is the possibility to express an algorithm in a way which comes closest to the structure of the underlying hardware of the machine, thereby making it possible to reach higher efficiency in the execution of a program. In contrast, when using a vectorizing compiler it is often difficult to understand why the compiler decides not to vectorize certain loops, which makes it hard to guess how to modify the code to trick the compiler into accepting those loops for vectorization.

A second advantage is the introduction of a more natural notation for vector operations. For example, vector addition can be written in many vector extensions of Fortran as

```
real a(N), b(N), c(N)
a = b + c
```

If only part of the arrays should be added, say from the *i*th to the *j*th element, the second line has to be replaced by

```
m = j - i + 1
a(i:m) = b(i:m) + c(i:m)
```

This notation is used for many compilers, and will also be used in Fortran 8X (on a Cyber, colons are replaced by semicolons).

The major drawback of current versions of vector-extended Fortran is the lack of compatibility among the different versions currently available on different machines. The introduction of Fortran8X hopefully will alleviate this problem. In addition, programs written with the vector notation of Fortran8X should run without modifications on scalar machines as well.

A second drawback of current vector dialects is their low-level character, bearing resemblence more to an assembly language than anything else. For example, the following piece of code which on a scalar machine reads as

```
do i = 1,k
   a(l(i)) = a(l(i)) + b(i)
enddo
```

is expressed in Cyber vector Fortran as

```
ac(1;k) = q8vgathr(a(1;x),l(1,k);ac(1;k))
ac(1;k) = ac(1;k) + b(1;k)
a(1;x) = q8vscatr(ac(1;k),l(1,k);a(1,x))
```

Instead, a higher-level way of expressing the same idea would be

```
a(l(1:k)) = a(l(1:k)) + b(1:k)
```

A realistic example of a more complicated piece of code, which occurs in the quadrupole version of the treecode discussed in section 3.3, reads on a scalar machine as:

```
  phiq = (-.5*((dx*dx - dz*dz)*
& quad(i,1) + (dy*dy - dz*dz)*
& quad(i,4)) - (dx*dy*quad(i,2) +
& dx*dz*quad(i,3) + dy*dz*
& quad(i,5)))*r5inv
```

On the Cyber, we had to replace this line by

```
  phiq(1;n) = (-.5*((dx(1;n)*dx(1;n) - dz(1;n)*dz(1;n))*
& quad(1,1;n) + (dy(1;n)*dy(1;n) - dz(1;n)*dz(1;n))*
& quad(1,4;n)) - (dx(1;n)*dy(1;n)*quad(1,2;n) +
& dx(1;n)*dz(1;n)*quad(1,3;n) + dy(1;n)*dz(1;n)*
& quad(1,5;n)))*r5inv(1;n)
```

There is obviously room for improvement in the notation. For example, it is clear that in one vector statement the length of all vectors involved should be the same. Thus a vector language should not require the user to specify the length for each vector separately. In addition, the notation would be less cumbersome if there would be a default interpretation for the starting address of a vector in the form of its first element. These two improvements would take away much of the assembly language flavor of the above expression, and instead would lead to something along the lines of

```
  phiq(1;n) = ( - .5 * ((dx * dx - dz * dz) *
& quad(,1;) + (dy * dy - dz * dz) *
& quad(,4;)) - (dx * dy * quad(,2;) +
& dx * dz * quad(,3;) + dy * dz *
& quad(,5;))) * r5inv
```

4.1.3. The C language of the Connection Machine*

The C* language [18] is developed by Thinking Machines Corporation, the company that manufactures the Connection Machine. Although C* is currently only implemented on the Connection Machine, it is a completely general parallel extension to C, and could equally well be implemented on vector machines as well as on other parallel architectures.

C* is a superset of C, and the data storage and computation on the frontend of the Connection Machine are controlled by the same syntax as in C. To store variables in parallel, in the local memory of the processor array, C* introduces the concept of a "domain", which has the identical syntax for declaration and referencing as the usual C "struct" construction for data structure. For example, the following code declares the basic data structure for an N-body system, and allocates N processors for that system:

```
domain particle{
  double position[3];
  double velocity[3];
  double mass;
}point[N];
```

Here is an example of a simple piece of parallel code, which updates the positions of all particles in an N-body system, simply by multiplying each velocity by a common constant small time step, and adding these values to the corresponding positions:

```
void particle::forward_euler_position(dt)
mono double dt;
{
   mono int k;
   for(k = 0;k < 3;k + + )}
      position[k] + = velocity[k]*dt;
}
void main()
{
   double dt;
   ...
   [domain particle].{
      ...
```

```
        forward_euler_position(dt);
        ...
    }
    ...
}
```

The domain is actually implemented as a class in the C + + language [19] which itself is an extension of C which has been available on scalar machines for several years. C + + is a superset of C which is designed to facilitate a more object-oriented style of programming. The declaration form for parallel functions in C* is the same as that for the member functions in C + +. The calling sequence is also similar. However, the reader does not need to be familiar with C + + to understand the C* language, although some knowledge about the former might make it somewhat easier to understand the latter.

In the first part of the code shown above, we declare that we use the structure *particle* as the target for parallel execution, and that we allocate N processors for that parallel target. In the parallel code, the parallel operations are expressed as operations on members of the domain, which are in this case *position, velocity*, and *mass*. All allocated processors have their own *position* etc. When the parallel code is executed, all processors operate on their own variables.

The keyword *mono* is used to indicate that a variable is not a parallel variable, but instead should be allocated in the main memory of the front end computer. By default, the variables declared in the body of the parallel functions are stored in the array processor and are therefore processed in parallel.

In the example above, the *for* loop is controlled by a *mono* (i.e. scalar) variable. This construction is not the only one allowed: a *poly* (i.e. parallel) variable can be used as well as control variable. This means that the number of iterations does not need to be equal for different processors. Actually we can write any control statement that is provided in standard C within the parallel code (even the *goto* statement will be supported!). This has the enormous advantage that we can write parallel pieces of code in the same style in which we are used to write our sequential programs. The only thing we need to add in order to execute such a piece of code in parallel is to add a declaration to that effect at the point where the functions are defined.

4.1.4. Comparison of the languages

When we use automatic DO loop vectorization of standard Fortran 77, we are forced to write every statement in such a way as to make sure that the compiler can recognize them as vectorizable. When instead using vector extensions of Fortran 77, we can write vector statements in a somewhat more natural way. The main drawback is the limited set of control structures in most vector-extended Fortran dialects. In addition, as discussed in the previous section, the notation used in the vector extensions is not very easy to read or to use.

In our opinion, the C* language is clearly superior in terms of flexibility and clarity. The main drawback is the lack of historical continuity, given the fact that most computer codes in physics have been written in Fortran. This problem is not relevant, though, when developing new codes for parallel computers. In fact, the transformation of codes from more traditional supercomputers to fine-grained parallel computers gives the physicist an ideal opportunity to step out of his nineteen-fifties based software straightjacket.

Let us summarize the main difference. (1) Supercomputers offer historical continuity by providing Fortran compilers. While useful in itself, this also introduces serious problems in writing an essentially parallel algorithm in standard Fortran in an automatically " vectorizable" form (cf. section 3), so that it can be executed effectively on supercomputers. (2) The Connection Machine currently lacks a Fortran compiler, although one will be provided in the future. However, it turned out to be fairly straightforward to write down our algorithms in C*, since C* is both a natural language in which to express parallelism as well as a strict superset of existing C. The only (temporary) drawback is that we were forced to modify our algorithms to work around current software and firmware limitations.

4.2. *Software environments*

4.2.1. *Vector processors*

In general, a supercomputer user has no direct access to the vector processors. Instead, she can communicate only with a front end computer, which is connected with the vector machine through a communication channel, and sometimes via shared secondary storage such as magnetic disks. Thus the interactive front end and the number-crunching back end form a loosely coupled network with the front end processor which takes charge of everything other than the execution of user programs.

In such an archaic, non-interactive setup the user cannot communicate directly with the vector processors. Instead, she first has to prepare her jobs on the front end computer, then submit it to a batch job on the back end computer, and finally wait for the vector processor to return some result. Only after inspection of the resulting data on the front end she would know whether the run has been succesful. To make matters worse, there are still a fair number of sites where the front end computer itself has an outdated type of mainframe operating system which is much less flexible than current workstation software.

The usual reason given for such an old-fashioned approach to computing has been the argument that supercomputers are mostly used for long-running number-crunching calculations, which take hours to finish. And indeed, a fully debugged and streamlined code may not require interactive access to a supercomputer in order to be applied efficiently. However, in developing such a code, interactive debugging sessions would be extremely helpful.

In addition, supercomputers could equally well be used to speed up calculations which can be done perfectly well on workstations overnight, but would take only a few minutes on a supercomputer. Being able to explore interactively a wide range of parameter space, for problems which would take many days of tedious bookkeeping of batch jobs on workstations, would be a second great advantage of interactive use.

Indeed, the first steps are now being taken to provide supercomputers with direct interactive access. For example, Cray is developing a version of UNIX, called UNICOS, which is already available on both their Cray-XMP and Cray-2 families. In addition, E.T.A, Fujitsu and N.E.C. have announced that they are developing versions of UNIX.

4.2.2. *The Connection Machine*

The Connection Machine, too, offers the user a front end through which to access the computer (either a Symbolics Lisp Machine or Vax 8800/8600 running Ultrix). However, the similarity with most supercomputers ends here. For the Connection Machine, the front end itself executes the program, and the parallel processor part of the machine mainly functions as "smart, active memory" for the front end. This has several advantages from the user point of view. First of all, the Connection Machine is fully interactive, because the user can directly log in on the front end. Secondly, a program running on the Connection Machine is actually executed on the front end, in the same way as a traditional program. Thirdly, it implies that any programming tool that is available on the front end can directly be used to develop programs for the Connection Machine system.

4.2.3. Comparison of software environments

The Connection Machine currently offers a much more interactive software environment than traditional supercomputers. Recently, however, some supercomputer vendors have begun to support some version of the UNIX operating system on their supercomputers. Even so, the software environments on supercomputers are likely to remain less flexible than the Connection Machine environment, mainly because of the last point mentioned in section 4.2.2: within the Connection Machine system *all* workstation tools are fully available. It is unlikely, for example, that a full implementation of Berkeley UNIX features will be available on any vector supercomputer in the foreseeable future.

5. Discussion

5.1. Results

We have evaluated the performance of the Connection Machine on two types of gravitational N-body algorithms, followed by an evaluation of the same algorithms on a variety of supercomputers. Our measurements reflect the state-of-the-art software of the Connection Machine during the fall of 1987, while the supercomputer benchmarks were performed during the spring and summer of 1988. Of the algorithms tested, the simpler type is based on a direct N^2 form of computing all gravitational interparticle interactions between N particles. The second, and considerably more complex type of algorithm is based on a hierarchical tree approach, in which the number of interactions calculated grows only in proportion to $N \log N$ [12].

We have found that tree algorithms are more efficient than direct N^2 type algorithms for particle numbers $N > 10^3 \sim 10^4$, both on the Connection Machine and on supercomputers. In contrast, on scalar machines such as workstations or mainframe, the turn-over point between the two types of algorithm is reached for particle numbers $N \approx 10^2$–10^3. This already forms an indication that the computing speed obtained for the tree algorithm is much lower than the peak speed, by an order of magnitude or more, for vector processors as well as for the Connection Machine.

Specific measurements of efficiency, defined as that fraction of the advertised top speed which could actually be obtained, gave the following results. On vector machines, the efficiency of the tree algorithm was in the range $5 \sim 15\%$. This speed was attained only after we modified the basic algorithm to a vectorizable form. The initial efficiency, without any modification of the scalar code, was typically $\sim 1\%$.

On the Connection Machine CM-2, we obtained an efficiency of $\sim 0.7\%$. This speed was again attained only after extensive modifications of the basic algorithm in order to work around the software limitations which were present at the time of our benchmarking. At the time this article appears in print, these limitations are expected to be largely remedied, and the predicted efficiency will then be $2 \sim 3\%$, or a factor $2 \sim 5$ lower than that of vector computers.

For both types of computers the reason for the low efficiency is the same, and can be traced back directly to the complicated pattern of communication in the tree algorithm. As a result, communication bottlenecks cannot be avoided, both on the Connection Machine and on supercomputers.

The absolute speed of the tree algorithm is, however, largely similar on the Connection Machine and on supercomputers, simply because of the larger raw speed of the Connection Machine which tends to cancel the lower efficiency (see fig. 6). Moreover, the price of the Connection Machine compares favorably with that of top-of-the-line vector computers. For small particle numbers, for which the simpler N^2 algorithm can be used, the Connection machine performs even better, reaching higher speeds than any of the vector computers which we have benchmarked (fig. 4).

5.2. Applications

Large-scale model calculations in astrophysics are among the most demanding types of computation in physics. The main reason is that gravity plays a dominant role on nearly any scale of interest for astrophysicists. As mentioned in the introduction, gravity introduces two extra complications which are not present in the more familiar hydrodynamics computations: mean free paths are longer than the size of the system, and gravity is a long-range force. In other words, if we would start with a simulation of a gas on the level of molecular dynamics, a transformation to the astrophysical N-body problem would eliminate the direct physical collisions and extend the range of interactions throughout the whole system. The first effect would imply that a particle could traverse the whole system on a dynamical time scale, while still retaining a good memory of its initial conditions, while the second effect implies that the particle–particle interactions are globally interrelated and highly time-dependent.

Another reason why astrophysical simulations are so demanding in terms of computer speed requirements is the essentially three-dimensional nature of many calculations. For example, a realistic simulation of colliding galaxies requires a fully 3-D approach, unless one would choose to study only the very special case of an exactly head-on encounter between galaxies with cylinder symmetry, oriented along the encounter axis.

Yet another complication which arises in almost any type of astrophysical modeling is the simultaneous occurrence of vastly different densities in different parts of the simulation. Whether one studies individual stars, gas clouds, or whole galaxies or clusters of galaxies, any self-gravitating system tends to develop strong density gradients. This is a significant difference with most laboratory experiments in physics, where a small portion of material is studied, often under near-homogeneous conditions. One example is convection, for which almost all results, theoretical as well as experimental, are obtained in the homogeneous approximation, a limitation which is most flagrantly contradicted in the interior of any star, and which has lead to a serious bottleneck in our understanding of stellar structure and evolution.

Similarly, in N-body experiments, self-gravitating aggregates of stars tend to spontaneously develop core–halo structures, which puts severe demands on the number of particles used in the simulation in order to simultaneously resolve all parts of the system, and on the range of individual time steps needed to resolve the different parts of the system in time as well as in space. For example, the encounter of two realistic disk-galaxies requires a minimum of several times 10^4 particles in order to provide sufficient resolution to follow the detailed evolution of the different components in each galaxy: their bulge, disk and halo [20].

The number of floating-point operations for such realistic encounters for disk galaxies can be estimated for a tree code as follows. (1) A single particle-node force calculation carries a computational cost of $30 \sim 70$ floating-point operations. These numbers follow from eq. (3.1), where a single acceleration contribution can be seen to involve fourteen floating-point operations, and three more operations to accumulate the total force (apart from a square root operation, the cost of which is highly machine-dependent; cf. table 2 and appendix A); a typical particle receives of order $\sim 10^3$ force terms arising from other particles and cells; (3) the number of particles in the system is $10^4 \sim 10^5$; (4) the number of total force calculations per particle per crossing time, which equals the number of time steps in integration schemes of the leap-frog and multi-step types, is a few times larger than the ratio of the size of the system and a typical interparticle distance, resulting in a number $\sim 10^{2.5}$; (5) the number of crossing times needed to describe a galactic encounter from beginning to end is of order $10^1 \sim 10^2$, depending on the desired degree of detail in the description of the encounter products. Multiplying all these numbers, we find a total cost of $\sim 10^{13}$ floating point operations per galactic encounter.

We conclude that a detailed study of galaxy encounters requires sustained access to a computer with a speed of about 100 Mflops, or alternatively periodic access to a computer in the Gigaflop range, in order to perform one scattering experiment per day. This requirement is dictated by the need to perform detailed parameter studies in which some of the many properties of the initial galaxies are changed in order to study their effect on the outcome of the collision. It is immediately clear that such types of investigation can only be carried out on supercomputers or high-performance parallel computers such as the Connection Machine. From the performance characteristics given in table 7, we can see that even the fastest supercomputers, as well as the Connection Machine, are currently barely able to provide the sustained 100 Mflops speed for our tree codes. Therefore, in a time-sharing environment, comprehensive parameter studies of realistic galaxy encounters will have to await the availability of computers faster than those benchmarked in the present paper.

In addition, many other problems in astrophysics are still well beyond present-day computational capabilities. To mention just one other example, the computation of a single evolutionary history of a globular cluster on a star-by-star basis requires a number of floating-point calculations of order 10^{19} [5,6,21]. Even if we have the patience to wait several months, and if other users do not revolt, we still need a machine of Teraflop speed. Although this exceeds the advertised speed of current computers by two orders of magnitude, such speeds are already within range of current hardware and software technology, as discussed below.

5.3. Conclusions

Our main conclusion is that the Connection Machine offers superior performance over vector supercomputers, on the level of net computational speed-per-dollar obtained for N-body codes of different types. In addition, the software environment on the Connection Machine system has the flexibility and interactivity of modern workstations, in contrast to most supercomputer systems. The main drawback of the Connection Machine at the time of writing of this paper (spring 1988), from the point of view of most physicists, is the current lack of a Fortran compiler. This implies that the Connection Machine offers significant advantages for those who want to develop new codes from scratch, as well as for those who have developed their codes in C or Lisp.

Older physics codes, which are generally available only in Fortran, may be more conveniently run on supercomputers, depending on the amount of labor involved in translating the codes from Fortran to C*. As soon as a parallel Fortran compiler will be available on the Connection Machine, however, the labor involved in a transition from a workstation to a Connection Machine is expected to be comparable to that of the transition from a workstation to a supercomputer.

An extrapolation of our results to the near future would suggest a very fruitful combination of the two types of technologies currently used in the Connection Machine and in vector supercomputers. When processors of supercomputer power would be connected in a fine-grained grid, using the type of software which already exists on the Connection Machine, Teraflop speeds are clearly within reach – without any novel development being required on either the hardware or the software side. Although the bottleneck would shift from the technological to the economical aspects of computer building, mass production of powerful computers would certainly make the price/performance of such "fine-grained supercomputers" far superior than that of either the current Connection Machine, or existing supercomputers.

Acknowledgements

We thank Joshua Barnes for useful discussions and critically reading the manuscript. We also thank Danny Hills, Brewster Kahle, Jacob Katzenelson, Gerald Sussman, and Feng Zhao for stimulating discussions concerning the Connection Machine.

We acknowledge the following institutions for providing us with computer time: the Computer Center of the University of Tokyo (Hitachi S-820); the Institute for Supercomputing Research (Cray X-MP; Facom VP-400); the John von Neumann National Supercomputer Center (Cyber 205); Thinking Machines Corporation (Connection Machine). We appreciate the hospitality of the above organizations, as well as the Institute for Advanced Study, the Massachusetts Institute for Technology, and the University of Tokyo, where parts of this research have been carried out.

Appendix A. Speeding up the square root

On some vector processors a square root calculation is surprisingly slow. Here we present a reasonably fast algorithm to replace the standard square root operation provided by the vendor.

The following iterative formula converges to r^{-1} quadratically:

$$a_{i+1} = \tfrac{3}{2}a_i - \tfrac{1}{2}a_i^3 r^2. \tag{A.1}$$

Table A.1

Machine	Gain in speed (%)
Hitac S-820/80	31
Facom VP-400	28

For this formula to converge, the initial guess a_0 should satisfy the condition $0 < a_0 < \sqrt{3}\, r^{-1}$. This scheme is faster than the usual Newton–Raphson formula to obtain r from r^2, because it does not require a division. Moreover, because what we need is r^{-1}, we can eliminate the division from the whole calculation. The main problem remaining is to obtain a reasonable initial guess. For the Newton–Raphson iteration the following expression gives a good starting value:

$$r_0 = \frac{|x| + |y| + |z|}{\sqrt{3}}. \tag{A.2}$$

The initial guess for r^{-1} is obtained by the following:

$$a_0 = \frac{2 r_0}{r_0^2 + r^2}. \tag{A.3}$$

Table A.1 gives the speed-up by the above scheme over the standard square root, for the direct summation algorithm we discussed in section 3.2.1. The speed-up is about 30% for both machines.

During the numerical integration of the gravitational N-body problem, we have a very good initial guess, namely the value of the distance at the previous time step. The one drawback of using the interparticle distances from the previous time step as initial guess is that this requires O(N^2) memory storage. The current vector processors provide a very large amount of memory of more than 10 Mwords. This is large enough to store all distances for a few thousands particles. By using this trick, we can eliminate the cost of the computation of the initial guess completely. The estimated speed-up by this technique is roughly an additional 30%. Unfortunately, we could not apply this technique to galaxy–galaxy scattering experiments. In that case, the time step is large. For particles that are very close to each other at one time step, the distance at that step is not a safe initial guess for the next step.

Appendix B. Example of C*

Here we present a non-trivial example of a C* program. We give two functions both of which form an implementation of the direct summation algorithm (b1) in section 3.2.2. The first one is written entirely in C*. The second one extensively utilizes PARIS (the assembly language for the Connection Machine) to obtain maximum speed.

Example of C* code – without using PARIS:

```
void node_domain::calc_accel_broadcast_all_c_star()
{
  int myindex;
  mono int i,k;
  REAL dx[NDIM];
  acc[0] = acc[1] = acc[2] = 0.0;      /* clear acceleration */
  potential = 0.0;                      /* clear potential */
  myindex = (int) this - (int) &node[0];
  /* obtain index of the processor itself */
  /* 'this' is the pre-defined variable */
  /* that has the processor's address */
  if(myindex >= 0 && myindex < nbody){/* deactivate unused processors */
    for (i = 0; i < nbody; i++){          /* loop over all particle */
      REAL rsq;
      REAL pot, rsqinv, rinv;
      if(myindex != i){                   /* avoid self interaction */
        rsq = eps2;
        for (k = 0; k < NDIM; k++){
          dx[k] = node[i].position[k] - position[k]; /* displacement */
          rsq += dx[k]*dx[k];                        /* r-squared */
        }
        rsqinv = 1.0/rsq;
        rinv = sqrt(rsqinv);
        pot = node[i].mass*rinv;
        potential += pot;                 /* accumulate potential */
        pot *= rsqinv;
        for (k = 0; k < NDIM; k++) { /* accumulate acceleration */
          acc[k] += pot*dx[k];
        }
      }
    }
  }
}
```

Example of C* code – using PARIS:

```
void node_domain::calc_accel_broadcast()
{
  int myindex;
  mono int i,k, mes_len;
  REAL dx[NDIM], dxdum[NDIM], host_work[NDIM + 1];
  REAL rsq, r_work[NDIM + 1];
  REAL pot, rsqinv, rinv;
  REAL r_work2[NDIM + 1];
  mes_len = (NDIM + 1) * REAL_LEN;
  acc[0] = acc[1] = acc[2] = 0.0;
  potential = 0.0;
  myindex(int) this – (int) &node[0];
  for (k = 0; k < NDIM; k + + ){
  r_work[k] = position[k];
  }
  r_work[NDIM] = mass;
  if(myindex > = 0 && myindex < nbody){
    for(i = 0; i < nbody; i + + ){
      if(myindex ! = i){
        CM_read_string_from_processor(host_work, &node[i],
                                            r_work,mes_len);
        CM_move_string_constant(r_work2, host_work, mes_len);
        rsq = eps2;
        CM_move(dx, r_work2, REAL_LEN * NDIM);
        for (k = 0; k < NDIM; k + + ){
          CM_f_subtract(&dx[k], &position[k], SIG_LEN, EXP_LEN);
          CM_move(&dxdum[k], &dx[k], REAL_LEN);
          CM_f_multiply(&dxdum[k], &dx[k], SIG_LEN, EXP_LEN);
          CM_f_add(&rsq, &dxdum[k], SIG_LEN, EXP_LEN);
        }
        CM_f_move_constant(&rsqinv, 1.0, SIG_LEN, EXP_LEN);
        CM_f_divide(&rsqinv, &rsq, SIG_LEN, EXP_LEN);
        CM_f_sqrt(&rinv, &rsqinv, SIG_LEN, EXP_LEN);
        CM_move(&pot, &r_work2[NDIM], REAL_LEN);
        CM_f_multiply(&pot, &rinv, SIG_LEN, EXP_LEN);
        CM_f_add(&potential, &pot, SIG_LEN, EXP_LEN);
        CM_f_multiply(&pot, &rsqinv, SIG_LEN, EXP_LEN);
        for (k = 0; k < NDIM; k + + ) {
          CM_f_multiply(&dx[k], &pot, SIG_LEN, EXP_LEN);
          CM_f_add(&acc[k], &dx[k], SIG_LEN, EXP_LEN);
        }
      }
    }
  }
}
```

References

[1] W.D. Hillis, The Connection Machine (MIT Press, Cambridge, 1985).
[2] W.D. Hillis and G.L. Steele, Commun. ACM 29 (1986) 1170.

[3] J.J. Monaghan, Computer Phys. Rep. 3 (1985) 71.
[4] S.J. Aarseth, in: Multiple Time Scales, eds. Brackhill and Cohen (Academic Press, New York, 1985), p. 377.
[5] J. Makino and P. Hut, Astrophys. J. Suppl. Ser. 68 (1988) 833.
[6] J. Makino and P. Hut (1989), in preparation.
[7] R.W. Hockney and J.W. Eastwood, Computer Simulation using Particles (McGraw–Hill, New York, 1981).
[8] P. Hut and S. McMillan, The Use of Supercomputers in Stellar Dynamics, (Springer, Berlin, 1986).
[9] A. Appel, SIAM J. Sci. Stat. Comput. 6 (1985) 85.
[10] J.G. Jernigan, in Dynamics of Star Clusters, I.A.U. Symp. 113, eds. J. Goodman and P. Hut (Reidel, Dordrecht 1985) p. 275.
[11] D. Porter, Ph.D. thesis, University of California, Berkeley (1985).
[12] J. Barnes and P. Hut, Nature 324 (1986) 446.
[13] L. Greengard and V. Rokhlin, J. Comput. Phys. 73 (1987) 325.
[14] F. Zhao, Master's thesis, MIT, Cambridge (1987).
[15] J. Barnes and P. Hut, Astrophys. J. Suppl. Ser. (1989), to appear.
[16] L. Hernquist, Astrophys. J. Suppl. Ser. 76 (1987) 64.
[17] J. Makino, J. Comput. Phys. (1989), to appear.
[18] J. Rose and G.L. Steele, preprint (1987).
[19] B. Stroustrup, The C+ + Programming Language (Addison–Wesley, Reading, 1986).
[20] J. Barnes, in: The Use of Supercomputers in Stellar Dynamics, eds. P. Hut and S. McMillan (Springer, Berlin, 1986), p. 175.
[21] P. Hut, J. Makino and S. McMillan, Nature 336 (1988) 31.