JOURNAL OF COMPUTATIONAL PHYSICS 111, 110–119 (1994)

The Development of a Parallel N-Body Code for the Edinburgh Concurrent Supercomputer

W. L. SWEATMAN*

Department of Mathematics and Statistics, University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh, EH9 3JZ, Scotland, United Kingdom

Received August 30, 1991; revised October 6, 1992

Sequential N-body codes have been in existence for many years. This paper shows how their algorithms may be adapted to produce an N-body code to run on a parallel machine and describes the implementation of such a code upon the Edinburgh Concurrent Supercomputer. The performance of the final program is analyzed, and an appendix discusses the optimal choice of the order of the integrator.

1. INTRODUCTION

N-body codes integrate through time the equations of motion for a collection of particles moving under forces between them. In the program developed during this study, the force involved is that due to gravity acting within a group of stars. Sophisticated algorithms have been previously written to integrate such models using serial computers [1 and references therein; 2]. The aim of the work described in the present paper was to write an N-body code that extended these techniques for use on a parallel machine: the Edinburgh Concurrent Supercomputer (ECS). The eventual program went through a succession of stages during development, as modifications were introduced to an initial version. The parallel framework of the code and the *master/slave* structure present in all the stages were suggested by an earlier N-body parallel code written by Duncan Roweth to run on the ECS. (This used the "leapfrog algorithm" which is explained later (Section 3.3.2).) The code developed has been used for systems in which the stars all have the same mass; however, with small modifications it would be ready for use in a multimass simulation.

This paper begins with a description of the computer environment. We go on to look at the program and its development, starting with the overall structure and then proceeding to study the development of particular parts of the program: the communications, computational algorithm, structure, and starting and finishing sequences. Having described the program, its performance is analyzed.

Throughout this paper we shall use the units of Heggie and Mathieu [3] unless otherwise stated. That is, we shall take $G = 1$, $M = 1$, and $E = -\frac{1}{4}$, where G is the gravitational constant, M is the total mass, and E is the total energy.

2. COMPUTING ENVIRONMENT

The ECS is an array of several hundred transputers. Each transputer contains a processor (rated at about 1 Mflop), memory, and communications. Each transputer has four "hard links," that is, connections that can be joined to other transputers to send messages between them. By making many such connections, transputers can be linked together to form large "concurrent" systems. The program whose development is described in this paper runs on such a system.

To program effectively in this environment, account must be taken of the time required to communicate between transputers; for computing efficiency it must be minimized. In the chain of communication it takes much longer to send messages between different transputers than internally between an individual transputer's processes (along "soft links"). In addition, one must avoid deadlock, the halting of one process to await input from another, when this second process in turn is awaiting further output from the first.

To use the full power of the machine, the dominant calculational part of the problem to be tackled must be divided into a number of nearly equal parts that can be solved independently on separate transputers. If the problem is not divided evenly, then some of the transputers will be left idle whilst others are still finishing off another part of the work.

* Current address: Department of Mathematics, Napier University, 219, Colinton Road, Edinburgh EH14 1DJ., Scotland, U.K.

0021-9991/94 $6.00

3. THE PROGRAM

3.1. Overall Structure

Right from the first version, the program could be divided into two sections: the *master* (one transputer) controls the system, initializes the calculation, collects the results, and is connected with input/output devices; the *slaves* (*s* transputers) perform the bulk of the calculations. Information about the stars needed for a particular transputer's calculations are stored in that transputer's memory in an array (called "*world*") with the successive data for different bodies arranged sequentially.

The program's operation is started by the *master*. It either reads in or generates a set of initial data for each star from which calculations can begin. It then communicates this data to the *slaves* and initiates the main sequence of operation. During this process the *slaves* numerically integrate the equations of motion of the stars, monitored and guided by the *master*. Once the stars have had their motion integrated to a preset time, the *master* terminates the main sequence of operation and, assisted by the *slaves*, finds any information required about the stars. This it lists in a file. A long integration is composed of several short runs during which the final data are listed at the end of each run and read into the next as initial data.

In the more detailed study of the program and its development which follows, we shall in succession consider its different constituent parts. We begin with the communication network between processors which remained more or less the same throughout program development (Section 3.2). Then we look at the algorithm used for the *slaves*' calculations, which underwent several changes (Section 3.3). Alongside these changes there were some adaptations made to parts of the larger program and these are detailed in Sections 3.4 and 3.5. The procedures used at the beginning and end of the run are commented upon in Sections 3.6 and 3.7, respectively.

3.2. Communications

The *master* and *slave* transputers are joined together in a simple loop (Fig. 1). Data are sent around this loop in one direction passing successively through the transputers. If the message is intended for the *master*, it is stopped there. Otherwise it is passed on by the transputers until terminated

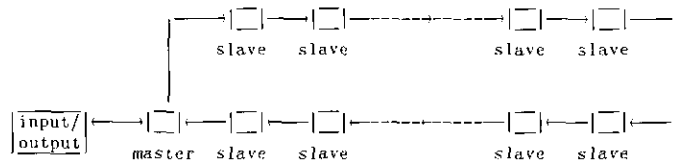

FIG. 1. The "hard links" providing communications between the transputers.

at the original sender, by which time it has passed through them all. The segments of program that run on each transputer are further divided into a main process that does the calculations and two buffer processes (Fig. 2).

The buffers facilitate communication between transputers. One buffer (*Inbuf*) collects messages arriving from the incoming hard link and passes these on to the main process. The other (*Outbuf*) collects messages from the main process and sends them out on the outgoing hard link. Together, by temporarily storing messages, they ensure that the hard links between processors are used efficiently and without "deadlocking." The *master* has an additional hard link running from the main process to the input/output devices.

Communication time could be improved, for instance, by refining the way in which the transputers are linked together. One could arrange the hard links into a branching structure to minimize the distance that messages have to travel between transputers. However, more significant gains are to be made by developing the calculation algorithms which tend to dominate the communications, and it is into this area that we proceed in the next section.

3.3. The Integration Algorithm in the Main Sequence of Operation

During the main sequence of operation the *slaves* numerically integrate through time the stars' equations of motion, under the *master*'s control. It is done by the repetition of a collection of operations. Consider a particular cycle. To begin with all the stars have their positions estimated at a time advanced from that of the previous cycle. Then, for a number of the stars, the force on them due to the others is calculated. This new data shall be used in later cycles' estimations of position. These stars are said to have been "updated"; they have had their equations of motion integrated to the time used in the cycle.

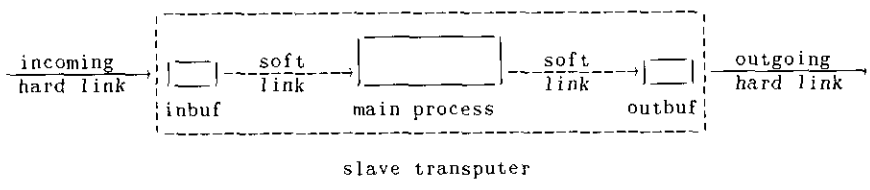

FIG. 2. A transputer's internal processes and communications.

To parallelize this section of operation the star updates are divided amongst the transputers: different *slaves* update different stars at the same time. As an illustration of such a cycle we shall outline the one contained within the final program, which had individual timesteps. The cycle begins with the *master* finding the next s stars to be updated (where s is the number of *slaves*). Then each *slave* updates one of these stars. Each *slave* repeats a loop: receive message from the *master*, stating which body to update; predict the positions of all the bodies and also the velocity of the body to be updated; calculate the new force on the body to be updated and compute its derivatives using divided differences (a method given in [2] and described in more detail in Section 3.3.5); add corrector terms to the updated body's predicted position and velocity and calculate the next time by which it must be updated; send the new data for the updated body to the other transputers and receive from them their new data on updated bodies.

Before we look at the initial algorithm used we should think about the statistical validity and accuracy of such algorithms.

3.3.1. *Statistical Validity and Accuracy*

We need to have some idea of the algorithm's statistical validity: does our solution resemble the exact solution starting from the same initial conditions? During an N-Body simulation, numerical errors accumulate and the system's coordinates deviate exponentially from the exact solution to the equations of motion and initial conditions. For typical codes, the error of the integrated model outstrips the accuracy of the computer in a few crossing times [4]. However, with a sufficiently high numerical accuracy it is hoped that the global properties of the simulation will resemble those of the exact solution (cf. [4, 5]). The deviation over the run of the total energy, an integral of the motion, is reckoned to be a good measure of the validity of these statistics.

The number of bodies and, to a lesser extent, the number of transputers influence accuracy; however, a more easily varied third influence is the size of the timestep (the difference in time between updates). For N-body codes, the timestep generally takes the form of a product $A\tau$, where τ is a time determined by local conditions and A is a constant. By varying A we alter the size of the timesteps and hence indirectly can adjust the error in the energy.

3.3.2. *Algorithm of the First Program: The Lockstep Leapfrog Algorithm*

The first working program used the "leapfrog" algorithm to integrate the motions of the stars through time. It is a lockstep algorithm; i.e., in each cycle all the particles have their positions and velocities updated to the same time. In fact in the program all the particles were updated together at constant intervals Δt. For each particle the cycle begins with the calculation of the force due to the other particles. Then the velocity is updated by the formula

$$\mathbf{v}_{\text{new}} = \mathbf{v}_{\text{old}} + \mathbf{f}\,\Delta t. \tag{1}$$

Finally, the position is given by

$$\mathbf{r}_{\text{new}} = \mathbf{r}_{\text{old}} + \mathbf{v}_{\text{new}}\,\Delta t. \tag{2}$$

(See Fig. 3.) $\mathbf{f}$ is the force per unit mass on the individual particle before the update, $\mathbf{v}$ is its velocity, and $\mathbf{r}$ is its position. The routine is repeated at every update.

The algorithm is called the leapfrog algorithm because in its most effective form the sequence of velocities for the bodies are taken to be at times differing by half a timestep from those of its positions (rather than concurrent with them). Then, the absolute truncation error in the energy per unit time is $O(\Delta t^2)$ (cf. [6]).

The algorithm is parallelized by dividing the star updates evenly amongst the *slaves*. We also only store the data for N/s bodies on each *slave* (although this will change during the program's subsequent development). (The number of stars in the system was chosen so as to be a multiple of the number of *slaves*.) After updating, the stars' velocities and positions are sent around the ring of transputers. They shall be used by each *slave* to find the new forces on its own N/s stars for the next update. A message contains data for all the stars stored on the transputer that sent it and hence its size is proportional to N/s. To go around the loop past all the *slaves* a message must travel across $s+1$ hard links. The communication time will be the product of these two factors; that is, approximately of order N. In comparison, the calculations made on each *slave* in between these communications are approximately $O(N^2/s)$, as on each *slave* there are N/s body updates each involving adding up the forces from $N-1$ other bodies. Comparing the order of the calculation and communication terms per body update, we see that the calculations are dominant for sufficiently many stars on each processor.

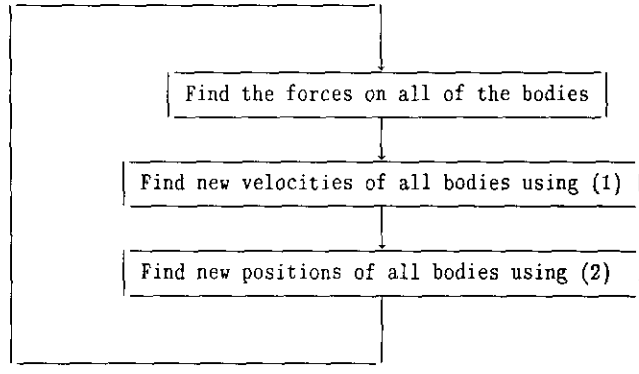

FIG. 3. A flow diagram for the leapfrog lockstep algorithm.

3.3.3. *Ideas for Algorithm Development from the Lockstep Leapfrog Program*

From the lockstep leapfrog algorithm a number of possible improvements may be made. In the final program we implement individual timesteps, variable timesteps, and a higher order in the prediction of orbits. These types of adaptations are explained by Aarseth [1] and customarily programs of this kind are called "Aarseth-type" after his famous "NBODY" series of codes. For the timestepping equations and numerical analysis for this kind of algorithm the reader is referred to [1, 2, 7, 8]. We have to modify these sequential techniques for optimal results on the parallel computer. Similar changes were also necessary for the earlier adaptations for vector computers as described by McMillan [9].

3.3.4. *Introducing Individual Variable Timesteps*

The first stage in developing the algorithms was the removal of lockstep and the introduction of individual, variable timesteps; that is the first two improvements mentioned in the last section were written into the program together.

Here a difference emerges between our program and standard sequential codes: the first modification (individual timesteps) is only partially implemented. To fit in with the overall structure (Section 3.1) bodies are not updated strictly individually but rather a fraction of them are updated at once, one star per slave. This enables us to obtain most of the gains in speed of strictly individual timesteps, whilst, at the same time, spreading the calculational load across the array and minimizing communication. The extra computation forced by premature update of bodies is studied in Section 4.1.

The first individual variable timestep used was a multiple of $r_{\min}/v_{r_{\min}}$, where $r_{\min}$ is the distance from the star in question to its nearest neighbor and $v_{r_{\min}}$ is the relative velocity of this nearest star. This timestep is comparatively easy to find—one can record $r_{\min}$ whilst performing the force calculations (during which the distances between the body being updated and all other stars are found), and then find $v_{r_{\min}}$ (using the predictor to calculate that nearest star's velocity at the present time). (This form of timestep might be improved in a mathematical sense, if it were instead the smallest ratio of relative distance apart divided by relative velocity, but the advantage gained has to be balanced with the extra calculation time required to find the relative velocity for all of the bodies.)

For each body, the time by which it must be updated along with the times of previous updates are recorded with the rest of its data in *world*. Now, as for the lockstep program, individual *slave* transputers store and perform the calculations for a subset of the bodies (N/s of them). As previously, there are partially overlapping calculation and communication phases; however, during the calculation phase only one body is updated on each *slave*, rather than all N/s. In order to update this body we require all the stars' positions at this time. They are in general calculated from truncated Taylor series ("force-polynomials") which are based on the estimates for position and its derivatives that were found at the time of the last update (t_{old}) (see Section 3.3.5). For this initial stage of program development, with just positions and velocities used, we are taking

$$\mathbf{r}(t) = \mathbf{r}_{\text{old}} + \mathbf{v}(t - t_{\text{old}}). \tag{3}$$

This process of predicting the positions of the bodies we shall call the extrapolations. At the end of the run all the bodies' positions and velocities are found by an extrapolation from their previous values.

With individual timesteps, there has to be a section of the program that determines which bodies require their orbit data updated next. After the body update, each *slave* uses a heapsort-like routine (see, e.g., [10, 11]) to find the star stored on it with the smallest time before the next update. This body shall be updated next upon the *slave*. The smallest times (one per *slave*) are sent to the *master*. A straight comparison of the s messages arriving at the *master* finds the smallest time of all the N bodies. This time is now sent around the loop of *slaves* so that they can use it to perform their next update.

3.3.5. *Changing to a Higher Order Algorithm*

After implementing individual variable timesteps the next progression made was to increase the order of the algorithm. Accompanying this change a new form of individual variable timestep was also introduced, which is described in Section 3.3.6. The new algorithm is shown as a flow chart in Fig. 4.

The algorithm used is a multistep integrator similar to that of NBODY1 [1] (but of one order lower in the final program). It is based upon the formulae given by [2]: a predictor–corrector scheme. The structure and properties of such algorithms are discussed and summarized in [8]. Essentially the predictor part of the method estimates the position of a star using a Newtonian extrapolating polynomial (a "force polynomial"). The predictor is used to find the position and velocity of a star whenever required in between the star's own updates (for other star's updates or at the end of a run). When it is time for a star itself to be updated, the predictor is used to find provisional values for its position and velocity. The predictors for other stars are also used to find their positions at this time. The force on the star being updated is calculated. The star's new acceleration, together with the previous acceleration values involved in its predictor, is used to obtain a new predictor and correc-

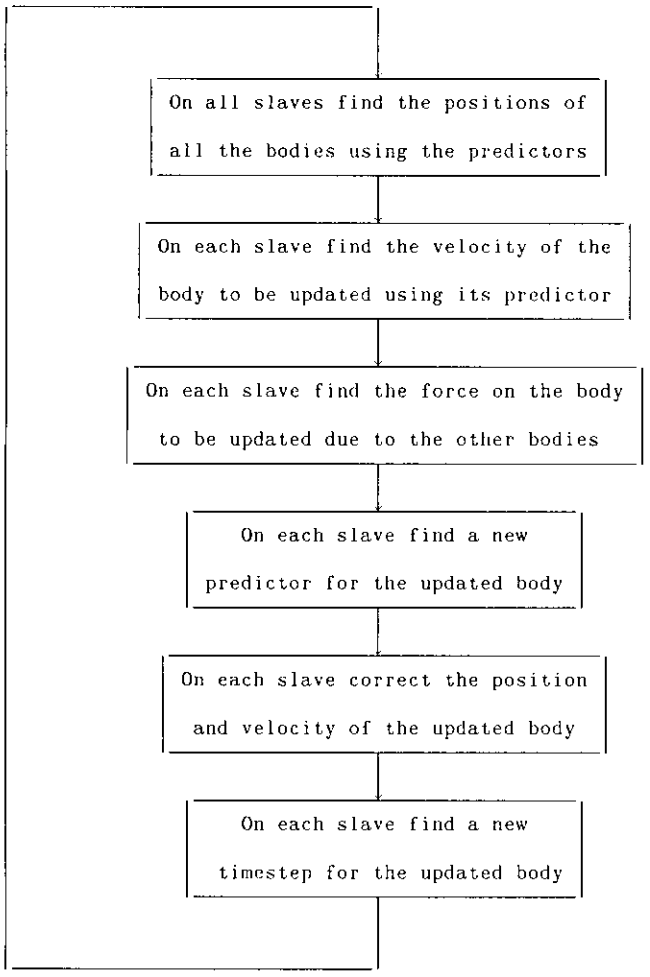

FIG. 4. A flow diagram for the higher order algorithm.

tions (of one order higher) to its provisional position and velocity. Finally, the next time of update for the star is found.

In practice we are only required to store the coefficients of the predictor and the update times for each star. (Each of the latter corresponds to a previous acceleration that is used in the predictor (there will be two less of these than the coefficients of the predictor as the order of the force polynomial is two less than that of the position).) When the previous accelerations of a star are required for use in its corrector and its next predictor, Wielen's formulae are rearranged to recover them from the coefficients of the current predictor. The program in its final form stored 20 items of data per star (in array *world*): the mass (taken to be $1/N$ throughout), five coefficients in each coordinate for the position's predictor, three previous update times, and the time of next update.

The correction terms improve the accuracy of the position and velocity at update; however, their primary use is as an estimate of the error before correction. We shall be using this in the construction of our new choice of timestep, which is described next.

3.3.6. *New Timestep*

To go with the higher order algorithm a more sophisticated formula is used for the timesteps. The individual timestep that is computed for each star is chosen to keep the estimated *absolute* error in energy per update within a bound (ε), preset by the user.

The energy of a single star (per unit mass) is

$$E = \tfrac{1}{2}\mathbf{v}^2 + \varphi(\mathbf{r}),$$

where φ is the potential at its position. So the error in this quantity is

$$\delta E = \mathbf{v}\cdot\delta\mathbf{v} + \delta\mathbf{r}\cdot\nabla\varphi(\mathbf{r}) \tag{4}$$

and

$$|\delta E| \leqslant |\mathbf{v}|\,|\delta\mathbf{v}| + |\delta\mathbf{r}|\,|\mathbf{f}|, \tag{5}$$

where $\mathbf{f}$ is acceleration. The errors in the position and velocity ($\delta\mathbf{r}$ and $\delta\mathbf{v}$, respectively) are taken to be the first terms that are not included in their respective Taylor series. For the timestep actually used in the update, these are precisely the correction terms mentioned in the preceding section.

It is reasonable to equate the error of a position or velocity series with the first truncated term. Press and Spergel [7] have shown that these terms will dominate higher order ones with a small enough timestep. In my program the actual truncation errors in position and velocity are of one order higher than their added-on corrections. However, we cannot find a timestep based on these error terms. Although we know their order we do not have any way of estimating their coefficients, and hence sizes, and it is these that we require for the timestep calculation. (There is an explanation in [11, pp. 555–556].) Therefore the correction terms themselves are taken as the truncation errors.

From (5) we obtain two conditions,

$$|\mathbf{v}|\,|\delta\mathbf{v}|,\ |\mathbf{f}|\,|\delta\mathbf{r}| \leqslant \varepsilon, \tag{6}$$

where ε is a preset constant: the energy error bound. We shall now show how to find a provisional timestep that is the largest one satisfying both these conditions. Upon completing an update we find two timesteps, Δt_1 and Δt_2. The first is the one that would have made the term $|\mathbf{v}|\,|\delta\mathbf{v}|$ equal to ε, and the second is that which would have made $|\delta\mathbf{r}|\,|\mathbf{f}|$ equal to ε. For these calculations we take $|\mathbf{v}|$ and $|\mathbf{f}|$ to be constant, assigning them their values at the actual update.

$|\delta \mathbf{v}|$ and $|\delta \mathbf{r}|$ are treated as being the first truncated terms from the Taylor series for velocity and position, respectively (constant multiples of powers of the respective timesteps Δt_1 and Δt_2). That is,

$$|\delta \mathbf{v}| = A\,\Delta t_1^n \qquad \text{and} \qquad |\delta \mathbf{r}| = B\,\Delta t_2^{n+1},$$

where n is the order of the predictor for estimating the position between updates. A and B may be found from the known values of $|\delta \mathbf{v}|$, $|\delta \mathbf{r}|$, and Δt at the performed update (the correction terms and actual timestep). The relationships between these quantities is

$$|\delta \mathbf{v}| = A\,\Delta t^n \qquad \text{and} \qquad |\delta \mathbf{r}| = B\,\Delta t^{n+1}.$$

Using (4), $|\delta E|$ would have been of the order of ε if we had taken Δt to be the minimum of Δt_1 and Δt_2, and so this value is provisionally taken to be the timestep to the next update.

In order to prevent the timestep from growing too fast, a further restriction is enforced. The provisional timestep is replaced by 1.4 times the previous timestep if the latter value is smaller (this stability factor is recommended by Aarseth [1]). So the new timestep finally takes the value

$$\Delta t = \min((\varepsilon(A\,|\mathbf{v}|)^{-1})^{1/n}, (\varepsilon(B\,|\mathbf{f}|)^{-1})^{1/(n+1)}, 1.4\,\Delta t_{\text{last}}). \quad (7)$$

For the eventual program $n = 4$ and this becomes

$$\Delta t = \min((\varepsilon(A\,|\mathbf{v}|)^{-1})^{1/4}, (\varepsilon(B\,|\mathbf{f}|)^{-1})^{1/5}, 1.4\,\Delta t_{\text{last}}).$$

(During simulations it was found that the timestep was hardly ever that associated with the term $|\delta \mathbf{r}|\,|\mathbf{f}|$; $|\mathbf{v}|\,|\delta \mathbf{v}|$ was generally the dominant term in the error of the energy.)

3.4. The Development of Structure from the Lockstep Leapfrog Program

In the lockstep leapfrog program the *slave* processors were only required to store data on a fraction of the stars, data on the rest being sent in from other processors as necessary. When body updates took place each *slave* predicted the position of N/s bodies (a process of order N/s per cycle) and then communicated this information to all the other *slaves* (a process of order N per cycle). As mentioned in Section 3.3.2 these processes were dominated by the force calculations for the body updates (which were a process of order N^2/s per cycle). However, after the introduction of individual, variable timesteps, only one body is updated in a cycle on each *slave* rather than N/s. This makes the force calculations per cycle of order N ($N-1$ gravitational forces are found between the body to be updated and the other stars). This is now of the same order as the communication (s updates are performed per cycle), and so communication time becomes important in the overall duration of the program operation. At this stage in the development, the program's speed was improved by a change in structure. Instead of having the *slaves* each only store a fraction of the stars' data, the new program had them each store it all. The *slaves* are made to duplicate each other's work by all *simultaneously predicting the positions of all the bodies.* This removes the necessity for the communication of the bodies' positions around the ring of transputers at the beginning of each cycle. The change does introduce extra work of order N per cycle (predicting N bodies' positions on each *slave*); however, the time taken to do this proves to be smaller than the aforementioned communication process (that is of the same order) which it is replacing.

3.5. Timestep Sorting on the Master

Following the previous change another simple improvement is to have the timestep sorting routine entirely on the *master*. This sorting is a comparatively small part of the computing time, being of order ($\log_2 N$) per body update. Timesteps for the updated bodies are sent to the *master* and there the smallest s of them are found using a heapsort-like routine similar to those which were formerly used on the *slaves*. By having the data for all of the bodies on all of the transputers we are able to update the coordinates of any body on any transputer. Hence we may update the s bodies with the smallest time to next integration, rather than the s bodies each of which have the smallest time on one of the *slaves* (the previous situation).

3.6. Initialization

The overall operation of the *master* and *slave* transputers during the main part of a run have been described. Before these routines can start, the system must be initialized: a set of data for the bodies must be generated or read from a file and sent to the appropriate transputers. To begin with, the initial conditions were generated by a subprogram that ran on the *master* transputer prior to the main code. Later, this subprogram was made into a separate program. At the start the initial positions and velocities of the stars in the model are set up using computer generated random numbers to fit a distribution function. Times for update and higher terms in the force polynomials are calculated from these initial positions and velocities as described in [1].

Contained within the more advanced codes for the *master* is a section to read in random initial conditions, or, equivalently, initial conditions that are the final data for a previous run. The *master* stores these data in the array *world* before sending them in sections to the *slaves*, around the loop. There are s sections (one for each *slave*); each section contains the initial data for N/s bodies. In the earlier programs each *slave* only recorded the information from

one section but in later programs they recorded the data for all of the bodies.

To end the initialization, the program finds the bodies to be updated and an initial timestep for them. This is done in the same way as in the main part of the code. Once this information has been broadcast to the transputers, the main part of the code's operation can begin.

3.7. The Duration of the Integration

Initially, when the timesteps were of a fixed length, a prescribed number of integrations were performed. With the introduction of variable timesteps the run was more directly limited by a preset time to which the equations were to be integrated. During the run the *master* monitors the time of the next update; when this exceeds the limit the *master* broadcasts a message that directs the *slaves* to cease their main operations. The current state of *world* is recorded for future runs, then the *slaves* update the positions and velocities to their values at the time limit and find any information required (e.g., the total kinetic and potential energies or the position of the center of the core.)

4. THE PERFORMANCE OF THE PROGRAM

During the development of the program, timed runs were performed at successive stages. A timer incorporated in the program recorded how long it took to update the system through a period of time. This excluded the time to set the program going and that to record the results at the end; the objective was to relate the parameters of the integration with the time spent doing the main calculations. The system also recorded the number of body updates that were required to perform the integration. In the next section, formulae are found for the time to update the system using the completed program and varying the numbers of bodies (N), and transputers. The program is compared with Aarseth's NBODY1 [1] running on the Edinburgh mainframe computer.

4.1. Tests Performed on the Final Program

The results presented here were obtained from a set of N identical masses whose initial positions and velocities were generated from a Plummer model (for a definition see, e.g., [12]). N was varied over a range of multiples of two between 16 and 2048, and the number of *slaves* between one and 128. The systems were all integrated through one of our time units which is $1/2\sqrt{2}$ crossing times. The energy error bound (ε) is taken to be $2^{-7} \times 10^{-2}$ for all these numerical experiments.

Empirically, for most of the results, the processing time per update (in seconds) is given to within 10% by the formula

$$\frac{1}{8000} \cdot \frac{N}{s} + \frac{7}{4000} \cdot s^{1/4}. \tag{8}$$

The two contributions are mainly due to computation and communication, respectively.

The first term, which is due to computation, is proportional to N/s. This is because for every body update the calculations are dominated by the prediction of the positions of all the bodies and the computation of their contributions to the force upon the body being updated. Both of these are of order N; however, s bodies are updated at once, one per *slave*, giving the N/s proportionality.

The power of $s^{1/4}$ on the second (communication) term is used purely because it is a good fit to the results. If the time to communicate between any two transputers in a system of any size was constant then this term, too, would be constant: there are s updates at once and the information from them is simultaneously passed around a loop of s transputers (if we ignore the *master*). In fact, as the number of transputers in the system grows, the communication time between them also grows and so the communication term is not independent of s.

Figure 5 shows the linear relationship between number of bodies (N) and processing time per body update, for various numbers of *slaves* (s). Also shown is Aarseth's NBODY1

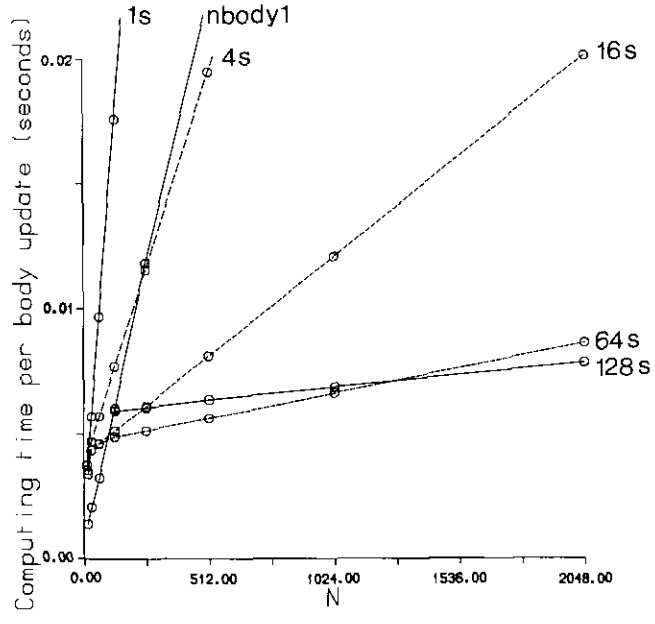

FIG. 5. The linear relationship between number of bodies in a stellar system (N) and computing time per body update, for various numbers of *slaves* (s). (The different numbers of *slaves* are represented by different lines as indicated (the line labelled 16s shows the results for an array with 16 *slave* transputers).) Also shown is Aarseth's NBODY1 running on a 2 Mflop scalar machine.

running on a 2 Mflop scalar machine. The different slopes of the graphs indicate the effect on the computational power of increasing the number of *slaves*, while the different y-intercepts indicate the presence of the second term in (8).

The main part of the calculational work done is $21N$ double precision multiplications per update (these are elaborated upon during the next section). So the computer is working at the rate $168s/(1+14N^{-1}s^{5/4})$ thousand double precision multiplications per second, giving for large N (when computation dominates communication) approximately 168,000 double precision multiplications per second per processor.

Using larger s the program takes less time per body update, but it also forces some updates sooner than are required by the timestep criterion. (For example, suppose that we have two stars which require update at times 0.010 and 0.015, respectively. If we update them simultaneously it must be to the time 0.010 and in this case the second star is updated 0.005 time units earlier than required. That star will be eventually updated more times than if it had been initially updated at 0.015 and only updated thereafter when strictly necessary.) Figure 6 shows how the number of body updates taken to integrate a system through a time unit relates to N for various s. As N increases the disadvantage of simultaneously updating s bodies gradually disappears. This graph indicates how much extra computation is done as we increase the number of *slaves* tackling a given N-body system. So, for instance, with 1000 bodies, four *slaves* do nearly the same amount of computation between them as one *slave* would have done; however, 64 *slaves* do over twice as much as either of these arrays.

Figure 7 shows graphs of the computing time to update the system through a time unit against N, again, for various s. For a given value of N the lowest line on these graphs will give the optimal (fastest) number of *slaves* to integrate the equations of motion, allowing for both extra body updates and longer interprocessor communication times.

The program has been tested in simulations of two Plummer-model systems with 1024 and 10,048 stars, respectively. Further details are given in [13]. For the 1024-body simulation the energy error bound was taken to be

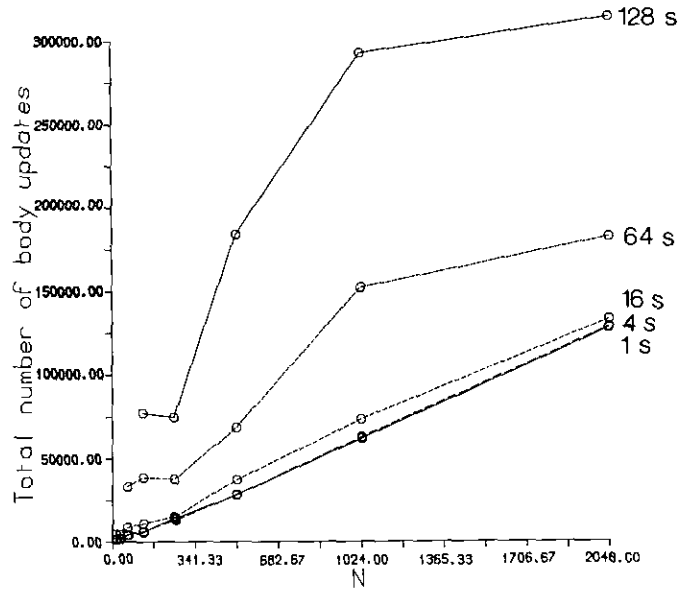

FIG. 6. The total number of body updates required to integrate a stellar system through one time unit plotted against number of bodies (N), for various numbers of *slaves* (s) (labelled as in Fig. 5).

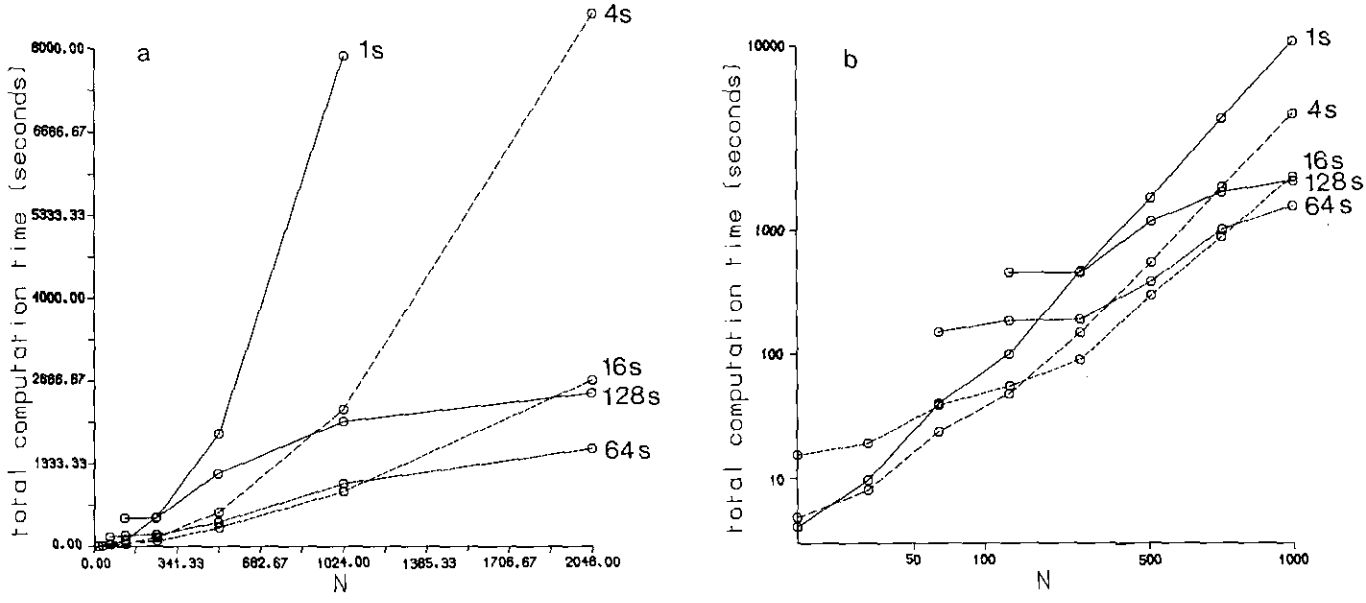

FIG. 7. The computing time to update a stellar system through one time unit plotted against number of bodies (N), for various numbers of slaves (s) (labelled as in Fig. 5): (a) linear scale; (b) log–log plot.

$2^{-7} \times 10^{-2}$. During 0.1 time intervals there were changes in total energy of nearly 2×10^{-5}; however, the deviation over 28.1 time units was less than 10^{-4} due to cancellation of these errors (cf. [8]). In order to maintain an energy error below 1.7×10^{-4} units per 0.1 time interval the energy error bound had to be reduced to $2^{-8} \times 10^{-2}$in the 10,048-body simulation.

APPENDIX: OPTIMAL ORDER

In programs of this type there is an optimal order for computational efficiency: the higher order schemes require an increasing number of calculations per body update so that there is a point beyond which the computer time involved in calculating the higher order terms balances out the saving in calculations due to having longer timesteps. The program developed here seemed to be reaching the point where only small gains were to be made by increasing the algorithm order any higher. Press and Spergel [7] study the choice of order in Aarseth-type N-body codes. We shall temporarily adopt their notation whilst studying the results of their paper. They define the extrapolatable interval (τ_E) from a time t_0 to be the maximum timestep such that an extrapolated value ($\hat{f}$) of the acceleration (found using the predictor part of the algorithm) has a fractional accuracy bounded by a limit (ε) for all smaller timesteps; i.e., τ_E satisfies the implicit equation

$$\frac{|\delta f|}{|f|} \approx \frac{|f(t_0+\tau) - \hat{f}(t_0+\tau, t_0, M, \tau)|}{|f(t_0+\tau)|} < \varepsilon, \qquad 0 \leqslant \tau \leqslant \tau_E.$$

M is the number of values of acceleration used in the predictor (one less than its own order). (We see that this is a criterion similar to that implied in Section 3.3.6. Total energy E is an integral of the system and so we may take that criterion to be $|\delta E|/|E| \leqslant \kappa$, where κ is a constant which is the energy error bound divided by the total energy.) Press and Spergel found that the ratio of the extrapolatable interval (τ_E) to a local timescale (τ_A) is typically constant to within a factor of two or three. They approximate the mean value of this ratio ($\langle s \rangle$) by a function of ε and M,

$$\langle s \rangle \approx 0.3 \left(\frac{\varepsilon}{0.07} \right)^{1/M}, \tag{9}$$

where $s = \tau_E/\tau_A$ and $\tau_A = |f/\ddot{f}|^{1/2}$.

They also give a formula for the optimal value of M for a given value of ε. It is arrived at by taking the force calculation as being dominated by evaluating the extrapolating polynomials, the time for which scales with M. Next the computation time is taken to be proportional to the force calculation time divided by $\langle s \rangle$, and this function is minimized to find the optimal M. In fact the calculation time scales as $(M + R)$, where R is a constant. This is because the force calculations for the updated particle are of the same order of magnitude as the extrapolations, but they do not depend on their order M. In my program R is four (cf. [6]).

Again differing from Press and Spergel, we assert that the optimal M should actually have a fixed error *per unit of time*. Therefore we take $\varepsilon/\langle s \rangle$, rather than ε (as used by Press and Spergel), as being constant. Call this constant ξ; then use of (9) gives

$$\langle s \rangle = \left(\frac{0.3^M \xi}{0.07} \right)^{1/(M-1)}. \tag{10}$$

The computer time to update the system through a fixed interval of its own time (T_M) is proportional to $(M + R)/\langle s \rangle$. If we regard this as being a function of M we may use it to determine the optimal M for a given ξ. We take the natural logarithm of the quantity and then set its derivative with respect to M to be zero. This gives the following formula to be satisfied by the optimal M:

$$(M-1)^2 + (M+R) \ln \left(\frac{0.3\xi}{0.07} \right) = 0.$$

Solving as a quadratic in $(M-1)$, we find that

$$M = 1 - \frac{1}{2} \ln \left(\frac{0.3\xi}{0.07} \right) \pm \frac{1}{2} \left(\left(\ln \left(\frac{0.3\xi}{0.07} \right) \right)^2 - 4(1+R) \ln \left(\frac{0.3\xi}{0.07} \right) \right)^{1/2}. \tag{11}$$

Taking $R = 4$, we must now estimate ξ. In NBODY1 when $M = 4$, a typical choice of timestep would be $[\eta(|f|\,|\ddot{f}| + |\dot{f}|^2)/(|\dot{f}|\,|\dddot{f}| + |\ddot{f}|^2)]^{1/2}$, where $\eta = 0.03$ [1]. Press and Spergel comment that the local timescale $[(|f|\,|\ddot{f}| + |\dot{f}|^2)/(|\dot{f}|\,|\dddot{f}| + |\ddot{f}|^2)]^{1/2}$ behaves very similarly to τ_A as given in (9) (an earlier timestep of Aarseth) and so we shall set $\langle s \rangle$ to be $\eta^{1/2}$. Now we can invert (10) to derive

$$\xi = \left(\frac{0.07}{0.3^4} \right) \eta^{3/2}. \tag{12}$$

We shall fix the value of ξ to obtain an error per time unit similar to that of NBODY1 with $\eta = 0.03$. Substitute (12) into the formula for optimal M (11) to obtain

$$M = 1 - \frac{1}{2} \ln \left(\frac{\eta^{3/2}}{0.3^3} \right) \pm \frac{1}{2} \left(\left(\ln \left(\frac{\eta^{3/2}}{0.3^3} \right) \right)^2 - 4(1+R) \ln \left(\frac{\eta^{3/2}}{0.3^3} \right) \right)^{1/2}. \tag{13}$$

Now $\ln(\eta^{1/2}/0.3)$ is negative, so we shall choose the positive root in order to make M positive. If we put in our value $R = 4$ we obtain

$$M \simeq 4.810.$$

Therefore we conclude that the optimal order for M with this error per time unit is 5 (or 4). Comparing the time to update the system through a fixed interval of time with $M = 5$ with that of another value of M, P, gives

$$\begin{aligned}\frac{T_P}{T_5} &= (P+4)\left(\frac{0.3^P\xi}{0.07}\right)^{-1/(P-1)}\frac{1}{9}\left(\frac{0.3^5\xi}{0.07}\right)^{1/4} \\ &= \frac{(P+4)}{9}\left(\frac{0.3\xi}{0.07}\right)^{1/4-1/(P-1)} \\ &= \frac{(P+4)}{9}\left(\frac{\sqrt{3}}{3}\right)^{3/4-3/(P-1)}.\end{aligned}$$

(Recall that $T_M \propto ((M+R)/\langle s\rangle)$.) The values of this ratio for P equal to 4 and 3 are approximately 1.020 and 1.174, respectively. The first result shows that the optimum M is indeed 5 rather than 4; however, it also indicates that NBODY1 would be barely improved by an increase in its order (at this accuracy). My final program has $M = 3$; as I was aiming for errors similar to those for NBODY1 as above, my algorithm should take less than $\frac{6}{5}$ the time taken with the optimal order algorithm.

Makino [8] has also studied the problem of optimal order. He differs with the results of Press and Spergel in that he asserts that the optimal order is dependent upon N. This may be true; however, his results are not appropriate for integrations at the accuracy of standard N-body codes in use. (His predictor of optimal order becomes negative unless the relative error in energy is very small for large numbers of bodies.)

ACKNOWLEDGMENTS

I thank Dr. Douglas C. Heggie for his helpful comments and encouragement with this work. A Science and Engineering Research Council grant helped fund this work, and computer facilities were provided by the Edinburgh Concurrent Supercomputer Project. I am also grateful to my three referees for their constructive criticisms on the preliminary version of this paper.

REFERENCES

1. S. J. Aarseth, in *Multiple Time Scales*, edited by J. U. Brackbill and P. I. Cohen (Academic Press, New York, 1985), p. 377.
2. R. Wielen, *Veroff. Astron. Rechen-Inst. Heidelberg*, Vol. 19 (Verlag Braun, Karlsruhe, 1967).
3. D. C. Heggie and R. D. Mathieu, in *The Use of Supercomputers in Stellar Dynamics*, edited by S. McMillan and P. Hut (Springer-Verlag, New York/Berlin, 1986), p. 233.
4. D. C. Heggie, *Predictability, Stability and Chaos in N-Body Dynamical Systems*, edited by A. E. Roy (Plenum, New York, 1991), p. 47.
5. D. C. Heggie, in *Long-Term Behavior of Natural and Artificial N-Body Systems*, edited by A. E. Roy (Kluwer, Dortrecht, p. 1988), p. 329.
6. W. L. Sweatman, Ph.D. thesis, University of Edinburgh, 1991.
7. W. H. Press and D. N. Spergel, *Astrophys. J.* **325** (1988), 715.
8. J. Makino, *Astrophys. J.* **369** (1991), 200.
9. S. McMillan, in *The Use of Supercomputers in Stellar Dynamics*, edited by P. Hut and S. McMillan (Springer-Verlag, New York/Berlin, 1986), p. 156.
10. W. H. Press, in *The Use of Supercomputers in Stellar Dynamics*, edited by P. Hut and S. McMillan (Springer-Verlag, New York/Berlin, 1986), p. 184.
11. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Receipes: The Art of Scientific Computing* (Cambridge Univ. Press, New York, 1986).
12. L. Spitzer, *Dynamical Evolution of Globular Star Clusters* (Princeton Univ. Press, Princeton, NJ, 1987).
13. W. L. Sweatman, A study of Lagrangian radii oscillations and core wandering using N-body simulations, *Monthly Notices R. Astron. Soc.* **261** (1993), 497.