

**Comparison of Parallel Solutions
to N-Body Problems**

M.Sc. in IT (Systems)

Seong-Gi SEO

1998-1999

Abstract

Five sets of codes dealing with N-Body Calculation are compared.

- Direct body-to-body serial code involving $O(N^2)$ computation.
- Barnes's tree-structure serial code involving $O(N*\log N)$ computation.
- Parallel version of direct body-to-body code.
- Parallel version of Barnes's code with static load distribution.
- Parallel version of Barnes's code with dynamic load balancing.

Conclusions are:

- Barnes's method is quite efficient incurring minimal overhead for the tree construction.
- In the direct body-to-body method, the computing time soon reaches the hardware limit as the number of bodies increases, showing its very limited applicability to only relatively small sized problems.
- The hierarchically structured computation method can deal with much bigger problems in much shorter execution time. On a cluster of 64/32MB RAM PCs, the time taken for the method to deal with a problem of 4096 bodies was an order of magnitude smaller with up to 3 processing elements.
- Even a simple, straightforward parallelisation of the two serial codes could produce good speedups. The parallel version of the body-to-body code achieved almost linear speedups. In the case of the parallel version of Barnes's code, it could not achieve a good speedup. However, this was not because of the inherent inflexibility in the algorithm, but because the tree construction part of the code has not been parallelised in this investigation.
- Dynamic load balancing is a necessity when the computational environments are changing with time to remove the load irregularity originated from the clustering. It is particularly true to achieve the optimum performance of the computing platform when using hierarchical tree-code because of the cluster formation of the bodies as the system evolves.
- The MPICH implementation of MPI is a very efficient message passing tool and very easy to use. Particularly, the collective operations such as `MPI_Allgather` and `MPI_Allgatherv` which were used in the main part seems to be very efficient – involving operations of $O(N*\log N)$ rather than $O(N^2)$ - and simple to use.

Table of Contents

Abstract.....	2
Table of Contents.....	3
List of Figures.....	4
Acknowledgement.....	5
1. Introduction.....	6
2. Theoretical Principles.....	8
2.1 Derivation of the Inverse Square Law of Force.....	8
2.2 N-Body Problem.....	10
2.3 Direct Body-to-Body Calculation.....	11
2.4 Barnes-Hut Algorithm.....	12
2.4.1 Barnes's Treecode.....	13
2.4.2 Barnes's Treecode Structure.....	14
2.4.3 Tree Structure.....	15
2.4.4 Tree Construction.....	15
2.4.5 Force Calculation.....	16
2.4.6 Time Integration.....	17
2.5 Parallelism in N-Body Problem.....	18
3. Software Design.....	19
3.1 Problem Analysis.....	19
3.2 Direct Body-to-Body Calculation.....	21
3.2 A Parallel Version of the Body-to-Body Code.....	22
3.4 Two Parallel Versions of Barnes's Code.....	23
3.4.1 A parallel version with static data allocation.....	24
3.4.2 A parallel version with dynamic load balancing.....	25
4. Implementation.....	26
4.1 The Computing Platform.....	26
4.2 MPI – A Message Passing Interface Standard.....	26
5. Results and Discussion.....	28
6. Conclusion.....	40
References.....	42
Appendices.....	Error! Bookmark not defined.
A.1 Running the Program.....	Error! Bookmark not defined.
A.2 Program Listings.....	Error! Bookmark not defined.
A.2.1 code.c.....	Error! Bookmark not defined.
A.2.2 n2code.c.....	Error! Bookmark not defined.
A.2.3 n2codell.c.....	Error! Bookmark not defined.
A.2.4 llv.c.....	Error! Bookmark not defined.
A.2.5 llv.c.....	Error! Bookmark not defined.
A.2.6 Common Files.....	Error! Bookmark not defined.

List of Figures

Fig 2.1 Acceleration of a circular trajectory	8
Fig.5.1 Evolution of the System	28
Fig.5.2 Comparison between N^2 Algorithm and $N*\log N$ Algorithm	29
Fig.5.3 Serial Version and Parallel Version of Barnes's Code	30
Fig.5.4 Serial Version and Parallel Version of Body-to-Body Code	30
Fig.5.5 Time Comparison of N^2 Algorithm and $N*\log N$ Algorithm	31
Fig.5.6 Speedups of N^2 and $N*\log N$	32
Fig.5.7 Times taken by the Parallel Body-to-Body Code	33
Fig.5.8 Speedup of Parallel body-to-Body Code	34
Fig.5.9 Times of Parallel Tree Code with Static Data Allocation	35
Fig.5.10 Speedup of Parallel Tree Code with Static Data Allocation	36
Fig.5.11 Parallel Tree Code with Dynamic Load Balancing	37
Fig.5.12 Speedups of Parallel Tree Code with Dynamic Load Balancing	38
Fig.5.13 Times for Parallel Codes with Static and Dynamic Load Allocation	39

Acknowledgement

I would like to thank Dr. Greg Michaelson and Professor Andrew Wallace for their support and guidance for this work.

I have been supported by the European Social Fund for the last 12 months and would like to thank the people who made it available to me.

1. Introduction

The problem dealt with in the present investigation was the N-body model of collisionless systems, in which the granularity of their mass distribution does not influence their evolution. The N-body problem deals with a group of bodies interacting with each other via forces that exist between two bodies. Only the gravitational force is considered. The gravitational force between two bodies is always attractive and defined by:

$$F_G = \frac{Gm_1m_2}{r^2}$$

where G is the universal gravitational constant of which the value is $6.673 \times 10^{-11} \text{Nm}^2/\text{Kg}^2$, m_1 and m_2 are the masses of the two bodies and r is the distance between them. The force acted upon a body by the rest of bodies in the system is the total sum of the force by each of the bodies. From the force thus calculated, the body's acceleration and then the new position after a certain time period can be calculated. The whole process is repeated at every time-step for the desired time span.

In a simulation process, the initial condition is usually set up by randomly distributing bodies with random masses and velocities. The system gradually evolves into clusters of bodies scattered about. This concentration of bodies can be viewed as a single body by a body located far away from it. This kind of approximation renders calculation methods which can be many orders of magnitude more efficient than the exact calculation method for very large problems.

Parallel processing usually involves domain decomposition. When the problem domain or the computing environment is changing with time, the static load distribution amongst the computing nodes may not be able to achieve the optimum performance of the system overall.

The augmented N-body problem is an example of dynamically changing data distribution. Here the bodies can appear and disappear at random, where the static domain decomposition may lead to a highly irregular load distribution among the processing elements.

A similar situation can arise when an N-body problem with constant number of bodies is handled in a hierarchically structured tree algorithm. Here a cluster of bodies can be approximated as a single body if the cluster can meet a predefined criterion. This in turn causes the amount of force computation for each body to be reduced – drastically, sometimes. Therefore the load on each processing element can vary significantly depending on the locations of the bodies each node has.

Since it is not possible to predetermine the amount of computation needed for each body and since the distribution of the bodies in the 3-D space is changing at each time step, it is necessary to have some means of dynamic load balancing as the computations are being carried out, to achieve the best efficiency of the computing system.

Barnes and Hut's algorithm is well known in the literature and Barnes's original hierarchical tree-code has been made available in the public domain by the author himself. This set of codes has been obtained through the Internet. To make the comparison as fair as possible, all codes were developed in exactly the same format. That is, the input/output method and format are identical, the same utility programs were used for calculations whenever possible and the same data structures were employed for the variables.

Altogether four more sets of codes have been developed. The first one used a direct body-to-body algorithm. In this algorithm the force calculation is made within a double iteration loop. It simply amasses all the induced effects by each body at a field point which is one of the bodies present in the problem domain. Thus the amount of computation involved is proportional to $(n*(n-1))$.

The second was the parallel version of the first. The simple approach taken to the parallelisation was to distribute an equal number of bodies to each and every processing node as the field points. This is, in practical terms, decomposing the outer loop of the double iteration and is better than distributing the source points. It is better because the granularity of the computation is coarser that way than decomposing the inner loop. The approach, incurring little overhead in the distribution process itself, was very effective and achieved almost linear speedups as expected.

The third and fourth sets of codes were parallel versions of Barnes's code, one with static load distribution and the other with dynamic load balancing. The tree construction part of the code was not parallelised due to the limited time for the current project. Only the data distribution part was parallelised. As a result, the speedup characteristics of these two sets of codes were not good when many processors were used in the computation. Nevertheless, the effect of parallelisation was evident – the computing time diminishes as more processors are engaged in the computation.

The main goal of the project was to study the scalability of the algorithms on coarse grained computing systems in a relatively architecture independent fashion. Distributed Memory Parallel Architecture is common nowadays. The message passing paradigm of programming is thus chosen. MPI, for its proven portability and efficiency, has been used in junction with C programming language. Therefore, the codes developed can be run on a wide variety of platforms including clusters of heterogeneous computer architectures. The codes were developed on a cluster of PCs with LINUX operating system installed.

2. Theoretical Principles

2.1 Derivation of the Inverse Square Law of Force

This section is a precis from [Bergmann(1969)].

According to Kepler's third law of planetary motion, the period of a planet is proportional to the $3/2$ power of the length of the largest diameter of its elliptic orbit.

On a circular trajectory, though the speed of the planet remains constant, its direction changes continuously. If the radius of the track is denoted by the symbol r , the circumference of the whole circle equals $2\pi r$. By the time the planet has traversed a small distance L the velocity, v , has changed by the small amount w , as depicted in Fig.2.1.

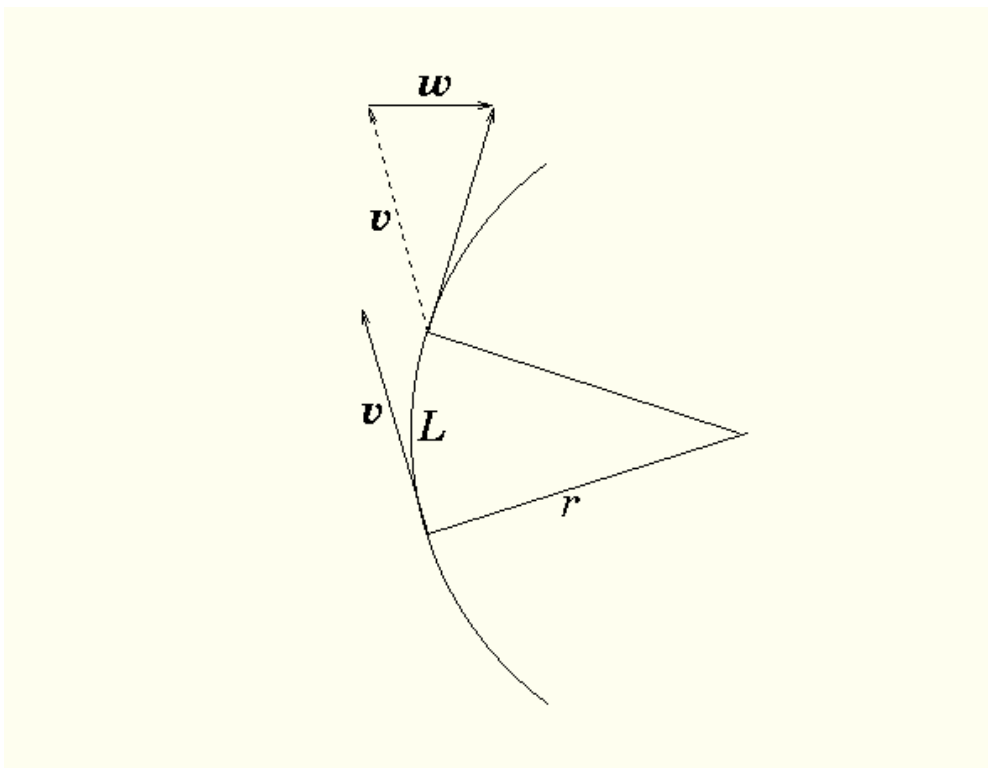


Fig 2.1 Acceleration of a circular trajectory

The ratio of w to v equals the ratio of L to the radius r , or **Error! Bookmark not defined.**

$$\frac{w}{v} = \frac{L}{r}$$

If the time required to traverse the distance L be denoted by t , and the time of one period by T , then the foregoing ratios amount to

$$\frac{w}{v} = 2\pi \frac{t}{T}$$

because the speed v equals the ratio of circumference to period, or $v = 2\pi r/T$. The ratio of w to t equals the rate at which the velocity changes per unit time, or the acceleration a , so that

$$a = \frac{w}{t} = 2\pi \frac{v}{T} = 4\pi^2 \frac{r}{T^2}.$$

As the force exerted on the planet by the sun equals the planet's acceleration times its mass, this force turns out to be

$$f = 4\pi^2 \frac{mr}{T^2}.$$

If the period T is proportional to the 3/2 power of the radius r , as established observationally by Kepler on the strength of the data collected by Tycho [Brahe](#),

Comment [PC1]:

$$T = b\sqrt{r^3},$$

substitution of this empirical law into the expression for the preceding force yields the desired dependence of the force on the distance of the planet from the sun:

$$f = \frac{4\pi^2}{b^2} \frac{m}{r^2}.$$

2.2 N-Body Problem

This section is a precis from [Pereira(1999)].

The N-Body problem deals with the motions of bodies that interact with each other. The state of the bodies – that is, position and velocity – is computed at each time-step through a set time period, starting from a given initial state.

In the gravitational N-body problem, each body is considered as a mass point characterised by a mass, a position and a velocity. The state of the system is defined by the set S_N of 3^*N parameters, namely, the masses, positions, and velocities of all bodies.

$$S_N = \{(m_i, r_i, \dot{r}_i), i = 1, 2, \dots, N\}$$

Where r_i and \dot{r}_i are the position and velocity vector of particle i , respectively.

The force exerted by particle j on particle i is given by Newton's Law of Gravity.

$$F_{ij} = -Gm_i m_j \frac{r_i - r_j}{\|r_i - r_j\|^3}$$

and the total force acting on particle i is

$$F_i = \sum_{j=1, j \neq i}^N F_{ij}$$

The right-hand side of the above equation represents the contribution of the other $N-1$ bodies to the total force.

Now the equation of motion of body i is

$$\ddot{r}_i = \frac{1}{m_i} F_{ij}$$

Defining $v_i = \dot{r}_i$,

$$\dot{v}_i = \frac{1}{m_i} F_i$$

with $i = 1, \dots, N$.

The evolution of the N-body system is determined by the solution of this system of differential equations with initial conditions.

2.3 Direct Body-to-Body Calculation

The simplest approach is to obtain the solution within a double iteration loop by calculating all forces induced by each of the bodies under consideration. Within each time-step the instantaneous acceleration is approximated by the instantaneous acceleration at the beginning of the time-step. This method is conceptually simple and vectorizes well, however, the amount of computation which is of $O(N^2)$ is usually considered as prohibitive for large-scale simulations involving millions of bodies.

Accelerations induced by the forces due to the $N-1$ bodies in the domain excluding the field point itself are calculated as

$$\frac{d^2 \vec{x}_i}{dt^2} = \sum_{j \neq i} \vec{a}_{ij} = \sum_{j \neq i} -\frac{Gm_j \vec{d}_{ij}}{|\vec{d}_{ij}|^3}$$

where, $\vec{d}_{ij} = \vec{r}_i - \vec{r}_j$ is the vector from the source point to the field point.

2.4 Barnes-Hut Algorithm

This section is a precis from [Barnes & Hut (1986)] and [Goil & Ranka (1995)].

Many physical systems exhibit a large range of scales in their information requirements in both time and space. A field point in physical domain requires progressively less information at a lesser frequency from source points in parts of the domain that are farther away from the field point. Applying this fundamental insight, Barnes and Hut proposed a faster N-Body algorithm involving computations of $O(N*\log N)$. The approximation made can be expressed as

$$\sum_j \frac{Gm_j \vec{d}_{ij}}{|d_{ij}|^3} \approx \frac{GM \vec{d}_{i,cm}}{d_{i,cm}^3}$$

where, $\vec{d}_{i,cm} = \vec{r}_i - \vec{r}_{cm}$ is the vector from the field point to the centre of the cluster which satisfies the approximation criterion.

In Barnes-Hut algorithm, the first stage to be done is the construction of tree structure. The bodies are inserted into the cluster hierarchy one at a time. An octree partition is computed recursively by dividing the original computational domain into eight octants of equal volume until each 3-D box contains only one body. The reduction in computation time is achieved by computing, for each field point, only the largest clusters which meet the approximation criterion. The bodies are added to the tree one at a time. The i^{th} Body is added into the tree in which $i-1$ bodies have already been inserted, the newly inserted body descending down the tree until it reaches a box in which it is the sole occupant. If the body reaches a leaf, the leaf is subdivided until each of the two bodies is in its own box. This tree building continues on in the same top-down fashion until all the bodies are positioned in its own compartment.

The second stage in BH algorithm is the calculation of attributes of each cluster, which is represented by each internal node. This is done traversing the tree from the bottom upwards. The attributes thus calculated now represent the group of bodies in the internal node, in other words, the whole subtree. The situation can be considered as the bodies collapsed to form a body at the centre of mass position.

The computation of accelerations of a body is carried out by examining each node in the tree in depth-first manner starting from the root cell. For each node the distance from the body and the size of cluster are compared to determine whether the node is located sufficiently far to be approximated as a point source, in which case the induced effects are calculated in the same way as body-to-body interaction.

The depth-first traversal here ensures that each body interacts only with the largest clusters which meet the approximation criterion.

Once accelerations on each body are known, the velocities can be calculated and with a given length of time-step the new positions can also be calculated.

The whole process is repeated for the desired number of time-steps.

The hierarchical “tree” codes are based on the idea of approximating the long-range force field of a localized region, containing many bodies, with some sort of multipole expansion. A tree-structure is used to partition the system into a hierarchy of such regions. To calculate the gravitational field strength of a point in the computational domain, a partial scan of the tree is first made to obtain a detailed description of the nearby mass distribution and an increasingly coarse-grained description of more distant parts of the system. If a better approximation is required, the algorithm may either examine a finer level of the hierarchy or include more moments of the mass distribution of each region.

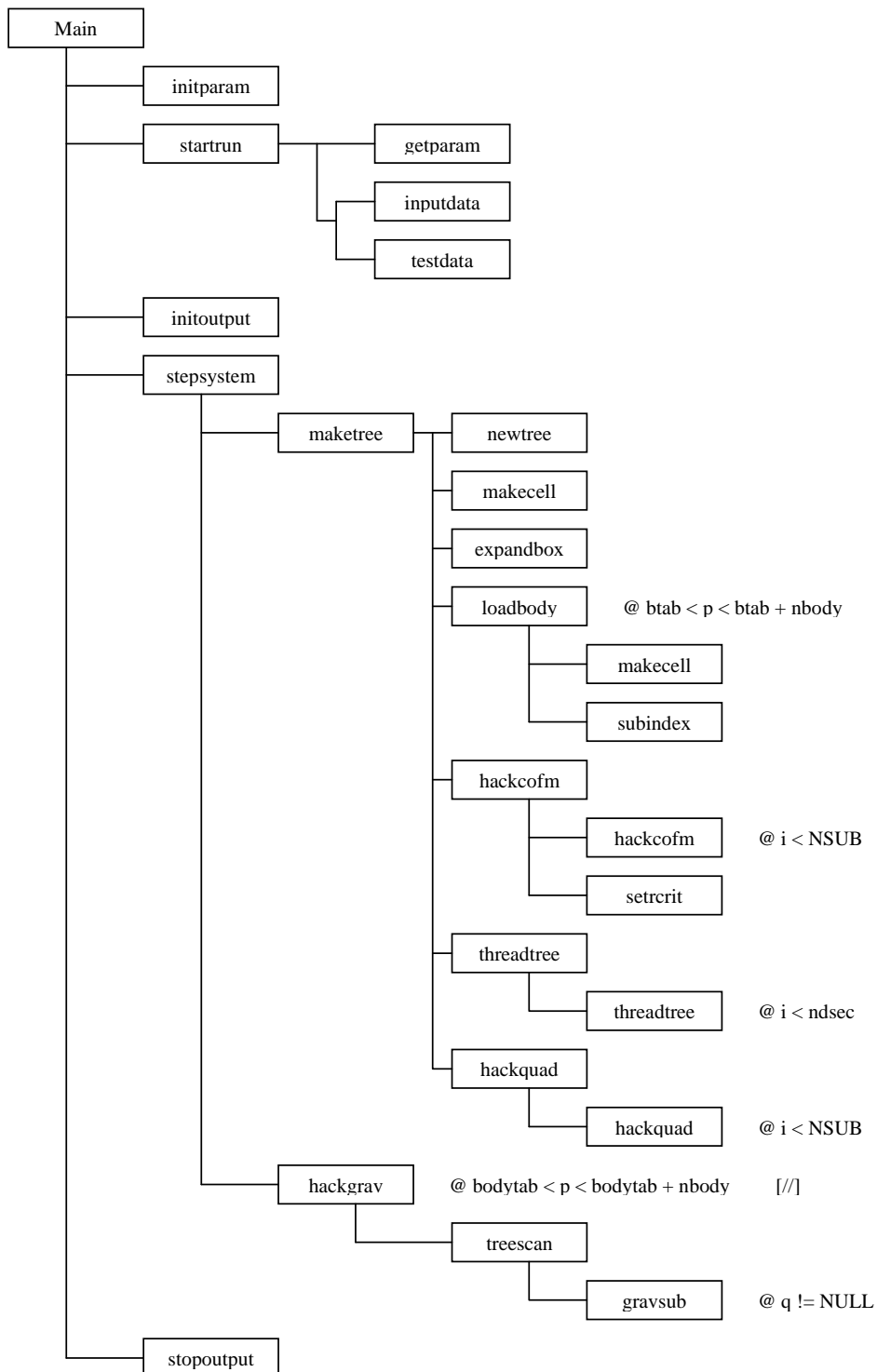
The strategy adopted to refine the force approximation is closely related to the representation adopted for the gravitational field. “Action-at-a-distance” codes represent the field at a given point r by a list of masses which together generate an approximation to the potential and force at r . This list is $O(\log N)$ elements long, so the time required to evaluate for the forces in all N bodies scales like $N \log N$.

2.4.1 Barnes’s Treecode

N-Body simulations using adaptive tree data structures are referred to as *treecode*. The main program of Barnes’s treecode calls, as an initial stage, a series of routines to initialise the system. With the data read in or set up, the code performs an initial force calculation, and outputs system diagnostics. The code then loops NSTEPS times, updating velocities and positions according to a simple time-centred leap-frog and re-computing forces from the new positions. The main loop is structured so that velocities and positions are synchronised at the beginning of each cycle. At regular intervals, and again at the end of the run, the body coordinates are output; the final output includes masses so that it may be used as initial conditions for a further run.

Initial conditions and parameter values may be specified in any set of units with $G \equiv 1$.

2.4.2 Barnes's Treecode Structure



2.4.3 Tree Structure

The next four sections including this is a precis from [Barnes].

A tree may be abstractly defined as a finite set T of nodes such that

- There is a single node r which is the root of T , and
- The other nodes, exclusive of r , are divided into $m \geq 0$ disjoint sets T_1, \dots, T_m , each of which is also a tree; T_i is the i^{th} sub-tree of T .

This definition is recursive – as are many of the algorithms used to manipulate trees- but it is not circular, since complex trees are defined in terms of simpler ones. The simplest trees consist of just one node; such a node is called a terminal node. Conversely, if T contains more than one node then its root is an interior node. Since every node is the root of some tree, every node is either terminal or interior.

Barnes uses “Eulerian” tree codes where the tree is constructed top-down by repeatedly subdividing a cubical cell enclosing the entire system, as shown in Fig. 3.1. In three dimensions, this generates an octonary tree structure, associating each cell with an interior node. A cell may be subdivided along the coordinate planes into eight smaller cells of equal volume; thus each cell has an ordered set of up to eight descending links, referred to as sub-cells.

The code generates a tree by subdividing the root cube until each body is isolated within a single cell. In practice, of course, it is redundant to allocate a cell to hold only one body, and such trivial cells are replaced with the bodies themselves. The number of non-trivial cells in the tree structure depends on the mass distribution, but is typically of order $0.5N$.

2.4.4 Tree Construction

The tree structure required for hierarchical force calculation is dynamic; it changes to reflect the mass distribution as the latter evolves. Rather than update the tree, this code simply regenerates it directly at each time-step.

Tree construction involves two phases. The first sets up the indices which link the tree together; the second computes total masses, centre-of-mass coordinates, critical radii, and quadrupole moments for all cells.

The function `loadbody` installs the body indexed by P into the tree structure attached to the ROOT cell. In doing so it also adds new cells as needed to insure that each body remain isolated in a single sub-cell. To find the correct place to insert P , this routine follows a trail from the root cell toward smaller cells, zeroing in on the single correct sub-cell.

If another body is already occupying the targeted sub-cell, a new cell must be added to the tree. The new cell is one-eighth the volume of the cell holding the original occupant; that is, it is exactly the volume of the contested sub-cell, and its midpoint is offset from its parent cell in the appropriate direction. Once the new cell is initialised, the old occupant is installed within it.

The function `hackcofm` performs the second phase of tree construction, computing total masses, centre-of-mass positions, critical radii, and quadrupole moments for all cells. This phase typically involves the flow of information from the leaves toward the root of the tree. It is therefore implemented with loops which scan the cells in order of increasing size; this insures that the sub-cells of any cell are processed before the cell itself is. Before the first such loop, cells are listed in order of decreasing size. The first loop computes total masses, centre-of-mass positions, and critical radii; if required, a second loop computes quadrupole moments.

The tree is reconstructed at each time step rather than undated. This was justified knowing that the tree construction takes only a small portion of the total computation time. In the case of 1024 bodies, for example, it takes less than 5% of the total time.

2.4.5 Force Calculation

Evaluating the force on a given body, p , is accomplished by a partial exploration of the tree, starting at the root. At each node q visited, a three-way decision is made. If q is a body, then the gravitational field due to q is added to the running total for p . If q is a cell, there are two further possibilities. If q is separated at sufficiently long distance from p , its gravitational field is again added to the running total. On the other hand, if q is too closed to p to yield an accurate force, the descendants of q must be examined instead.

A cell is deemed far enough away from p to be approximated as a single mass if

$$\frac{l_q}{\theta} \langle |\Delta \mathbf{r}| \rangle \quad (2.1)$$

where l_q is the linear size of q and δ_q is the distance between q 's centre-of-mass and its geometric centre. $\Delta \mathbf{r} \equiv \mathbf{r}_q - \mathbf{r}_p$ is the vector extending from p to the centre-of-mass of q , and θ is a user specified parameter used to control the accuracy. Since the critical radius $l_q/\theta + \delta_q$ depends only on the parameters of the cell q , it is conveniently evaluated during tree construction, and the result is stored in the data structure associated with each cell for quick reference during force calculation. Eq. (2.1) positively guarantees $p \notin q$ for any $\theta < 2 / 1.7321 \cong 1.155$.

The contribution of a gibe node q to the gravitational field at body p depends on the type of q . If q is a body, its contribution to the potential at p is simply

$$-\phi_q = G \frac{m_q}{(|\Delta \mathbf{r}|^2 + \epsilon^2)^{1/2}}. \quad (2.2)$$

where m_q is the mass of q . On the other hand, if q is a cell, its contribution to the potential may also include a quadrupole correction, giving

$$-\phi_q = G \frac{m_q}{(|\Delta \mathbf{r}|^2 + \epsilon^2)^{1/2}} + G \frac{\Delta \mathbf{r} \cdot \mathbf{Q}_q \Delta \mathbf{r}}{2(|\Delta \mathbf{r}|^2 + \epsilon^2)^{5/2}} \quad (2.3)$$

where \mathbf{Q}_q is the traceless quadrupole moment of the cell q . In either case, q 's contribution to the force on p is obtained by symbolically differentiating ϕ_q with respect to \mathbf{r}_p .

To compute the force on a body, the code makes a partial traverse of the tree structure. The goal is not to visit every node; if a sufficiently accurate approximation to the force can be obtained from a given cell, there is no need to visit its descendants.

A node is examined to see if it is a cell and if it is located far enough to be regarded as a single body. If not far enough, it is sub-divided and the pointer is stored. If it is a body, the gravity is computed. The process is repeated for each and every body in the system.

2.4.6 Time Integration

In collisionless N-body models the choice of time-step is tied to issues of global stability, and it is not entirely clear how to go about selecting individual time-steps. In the interests of simplicity, the code therefore uses a time-centred leap-frog, advancing all bodies with the same time-step parameter Δt chosen by the user. This approach is simpler, more robust, and more economical of memory than available high-order schemes.

As normally formulated, the leap-frog requires that velocities be offset by half a time-step with respect position. However, the method can easily be recast as a mapping from time $t^{[n]}$ to time $t^{[n+1]} = t^{[n]} + \Delta t$ (e.g., Barnes & Hut 1989). Let $\mathbf{r}^{[n]}$ and $\mathbf{v}^{[n]}$ be position and velocity of a body at time-step n ; the body is advanced as follows:

$$\begin{aligned} \mathbf{V}^{[n+1/2]} &= \mathbf{V}^{[n]} + \frac{1}{2} \Delta t \mathbf{a}(\mathbf{r}^{[n]}) \\ \mathbf{r}^{[n+1]} &= \mathbf{r}^{[n]} + \Delta t \mathbf{V}^{[n+1/2]} \\ \mathbf{V}^{[n+1]} &= \mathbf{V}^{[n+1/2]} + \frac{1}{2} \Delta t \mathbf{a}(\mathbf{r}^{[n+1]}) \end{aligned} \quad (2.4)$$

where $\mathbf{a}(\mathbf{r})$, the acceleration obtained from the force calculation, depends on the positions of all bodies. This reformulation does not compromise the desirable properties of the leap-frog integrator.

2.5 Parallelism in N-Body Problem

The main part of N-Body problem is the calculation of force and the consequential quantities of a body, due to the rest of bodies in the computational domain. Since the locations and masses of the bodies are assumed to be fixed at the beginning of each time-step, the force due to a body is constant, that is, independent of the rest of bodies present. This renders an ideal opportunity for parallelisation, particularly when a large number of bodies are treated. This is because the architectures commonly available nowadays have many processing elements each of which is fairly powerful and has fair amount of local memory, thus are best for coarse grained parallelism at program level.

In a straightforward body-to-body algorithm, the obvious way to parallelism is allocating an equal (average) number of bodies (field points) to each processing element. In the tree-structured algorithm, however, the numbers for individual processing elements should be chosen so that each of the element may have an approximately equal amount of computation load. In the latter case, the construction of tree should also be considered to be shared among the PEs. The cost for the tree construction is only a few percent in a sequential algorithm. If, however, hundreds, or thousands possibly, of PEs are used for the computation, the percentage of tree construction will far outweigh that of force calculation in each PE.

In view of Amdahl's Law, even a small part of the program could cause serious performance degradation if left as serial code, particularly when many processors are sharing the computation load.

3. Software Design

3.1 Problem Analysis

The necessary statistics for the analysis were obtained by using the software tools for profiling such as 'prof' or 'gprof' and by implanting some timing or counting routines within the code.

The first task was to find out how expensive the tree building is. This was done by incorporating the `-p` option for profiling when the source code (Barnes's `treecode`) was compiled. The following is an example when 1024 bodies were used.

%Time	Seconds	Cumsecs	#Calls	msec/call	Name
66.2	14.52	14.52	66560	0.2181	hackgrav
10.6	2.33	16.85	66883	0.0348	_times
10.3	2.25	19.10			_mcount
8.6	1.89	20.9913362008		0.0001	__sqrt
1.5	0.32	21.31	65	4.9	maketree
1.0	0.22	21.53	65	3.4	output

The above information says that the call to 'maketree' costs only 1.5% of the total execution time. 'gprof' provides much more detailed information on every user and system routines. It was finally concluded that the cost of tree-building was less than 5% of the total execution time. This was in harmony with the observation by Barnes himself. [Barnes] Barnes used this fact as the justification for rebuilding the tree anew for at each time-step rather than updating it.

The results of timings done by inserting UNIX or C timing functions also indicated similar statistics.

Due to the limited time for the course and the complexity involved in parallelising the tree-building part of the code, it was decided that the task be left for future work. Only the force calculation part which accounts for most of the rest of cost was parallelised. It was fully recognised even at this stage that the cost of tree-building would take increasingly more portion of the total cost as the job was shared by increasing number of processing elements.

The second aspect to be investigated was how efficient the static load distribution was. To find out the answer, the frequency of force calculation for each body was counted. It turned out that the counts vary vastly from 13 up to 282 in the case of 1024 randomly distributed bodies. This means that the 'treewalk' goes down deep for some

bodies and for some others very shallow. The irregularity will become worse as the time-step goes on because the bodies will form clusters as they evolve. This fact clearly indicates the need to have some kind of dynamic load balancing.

3.2 Direct Body-to-Body Calculation

The program was written within the framework of Barnes's code. [Barnes 1993] The core calculation part (nsquared) was newly written and replaced 'stepsystem' which is for building the tree and force calculation in Barnes's tree-code.

In his original code, Barnes organised for the velocities to be calculated in two stages each for half Δt . This no doubt will increase the computing time unnecessarily. Thus this part of program was reorganised so that leap-frog can happen every Δt except when NSTEP=0. This resulted in some performance improvement. The very initial leap-frog at NSTEP=0 was done by 'nshalfdt' which was also newly added.

The outputs are only the positions of bodies at the end of each time-step, which is the input data for the MATLAB plot file, plot6.m. Eleven points at regular interval were selected for the plot file from the results of 1024 body case. The pseudo-code below shows the overall structure of the code.

```

/* MAIN: toplevel routine for hierarchical N-body code. */
initparam(argv, defv);      /* setup parameter access */
startrun();                 /* set params, input data */
while (tnow < tstop) {      /* while not past tstop */
    nsquared();
    output (coordinates); }

/* nsquared : direct body-to-body calculation */
if (nstep == 0) {
    for (i=0; i<nbody; i++) {
        for (j=0; j<nbody; j++) {
            obtain vector from source to field point
            calculate acceleration using Newton's Law } }
    for (i=0; i<nbody; i++) {
        calculate velocity change for half time-step
        update velocity
        calculate displacement for one time-step
        obtain new position } }

    for (i=0; i<nbody; i++) {
        for (j=0; j<nbody; j++) {
            obtain vector from source to field point
            calculate acceleration using Newton's Law } }
    for (i=0; i<nbody; i++) {
        calculate velocity change for half time-step
        update velocity
        calculate displacement for one time-step
        obtain new position }
    for (i=0; i<nbody; i+=100) print out new positions
    nstep++;                  /* count another time step */
    tnow = tnow + dt;

```

3.2 A Parallel Version of the Body-to-Body Code

In the body-to-body code, the basic unit of computation is that of the force and the consequential displacement caused to a field point by a source point. This basic unit is iterated in a double loop for each and every body. Each basic unit of computation is completely independent of the others. Therefore, either the inner loop or the outer loop can be decomposed for parallelisation. Since the amount of computation for the basic unit is very small for modern processing elements, fine-grained parallelism may underuse the processors and suffer from the relatively bigger communication overhead. Thus the outer iteration loop was parallelised, which practically means that the source bodies are distributed amongst the available processing elements.

Barnes used a particular data structure for velocities and positions, which were difficult to use with MPI. It was thus decided to pack and unpack the data in the MPI defined data type for the communication. Packing the data was done by 'packvp' and unpacking was done by 'buftovp'. Both of the routines were newly added. In the code, as soon as new velocities and positions are available, they are passed around. The collective communication was done by MPI_Allgather which involves work of $O(N \log N)$ in contrast to $O(N^2)$ which is necessary if it is done individually. The following pseudo-code shows the overall structure of the code.

```

/* MAIN: toplevel routine for hierarchical N-body code */
MPI initialisation
initparam(argv, defv);          /* setup parameter access */
starttr();                      /* set params, input data */
n2halfdt;                       /* initial half step */
packvp(bufv, bufp);             /* pack new values into buf */
MPI_Allgather(bufv, rbufv);     /* new velocities */
MPI_Allgather(bufp, rbufp);     /* new positions */
buftovp(rbufv, rbufp);         /* unpack new values */
while (tnow < tstop) {         /* while not past tstop */
    n2dt;                       /* one full time step */
    packvp(bufv, bufp);
    MPI_Allgather(bufv, rbufv);
    MPI_Allgather(bufp, rbufp);
    buftovp(rbufv, rbufp);
}
MPI_Finalize();

```

3.4 Two Parallel Versions of Barnes's Code

Although the bodies are initially distributed randomly, that is, uniformly, through the evolution there forms clusters. The tree now can be highly imbalanced. That is, some of the branches can be very long, whereas others very short. B-H algorithm deals with the clusters as a single body if it meets the predefined criterion, which implies that the amount of calculation for each body will vary depending on its location relative to the clusters. An efficient algorithm should take this into consideration and try to assign uniform amount of load for each processing element.

New positions of bodies are calculated in "stepsystem". The accumulated values of $Acc(p)$ and $\Phi(p)$ are finally stored in subroutine "hackgrav". Hackgrav is called once for each body. Hackgrav calls "treescan" once and treescan calls "gravsub" different number of times depending on how deep in the tree it goes down. Therefore, for each body the amount of calculation, that is the time required, for the field effects on the body itself is all different.

If the number of calls for gravsub can be made (almost) uniform for each processor, the computing hardware system would be working at its optimum efficiency. "treescan" itself does not do much calculation, but it decide how often gravsub should be called. A simplified version of treescan could be used just to obtain the information on the number of calls for gravsub, which can then be used to uniformly distribute the load among the processors before the call of the actual treescan. If the simplified version of treescan takes significant amount of time, a way round should be devised. Unfortunately this proved to be the case. In the case of 1024 bodies, it caused nearly 30% overhead, dictating the need to find an alternative.

The actual number of calls made to gravsub in the previous time step can be used for the load balancing at the next time step as an ad hoc approximation. The fact that each body does not change its position drastically as long as the time step is kept short enough can be the justification for the ad hoc method.

The first modification made to the tree codes was related to the linear approximation of the bodies' movement. In his original code, Barnes separated the process into two and two separate velocity calculations are done each for half time step. In the modified code, the two are combined together, which resulted in some significant saving in execution time – typically 20-30 %.

3.4.1 A parallel version with static data allocation

Since it was decided that the tree construction part be left sequential for the time being, only the force calculation part was parallelised. Both of the functions were inside the 'stepsystem'. Thus the calls to 'hackgrav' within 'stepsystem' were distributed amongst the processors. The pseudo-code below shows the way it was done.

```

/* STEPSYSTEM: advance N-body system one time-step.*/

if (nstep == 0) {
  build tree structure
  for (i=istart; i<=iend; i++) {          /*loop over allocated bodies*/
    get force on each body
  } }

for (i=istart; i<=iend; i++) {          /*loop over allocated bodies*/
  get velocity increment during half a time step
  update velocities
  get positon increment
  advance r by one step
}

packvp(bufv, bufp);                    /*pack v&p for communication*/
global update of velocities and positions
buftovp(rbufv, rbufp);                /*unpack v&p          */

build tree structure
for (i=istart; i<=iend; i++) {          /*loop over allocated bodies*/
  get force on each body
}

for (i=istart; i<=iend; i++) {          /*loop over allocated bodies*/
  get velocity increment for half a time step
  update velocities for half a time step
}

for (i=0; i<nbody; i+=100) {
  output new positions of every 100 bodies
}

nstep++;                               /* count another time step */
tnow = tnow + dt;                       /* finally, advance time   */

```

3.4.2 A parallel version with dynamic load balancing

At each time step the amount of force calculation for each element is counted and this information is used to dynamically assign approximately uniform load for each processing element for the next time step. This process incurs a little overhead but the benefit will soon overcome it reducing the overall processing time.

The total number of calls to “qsub” is divided by the number of processors giving the average number. The average is the smallest integer number which is not less than the rational average. The processor whose rank is 0 is given bodies whose cumulative number of 'qsub' calls is not less than the average, starting from body 0. Rank 1 is given bodies from the next up to the one up to whose cumulative call number from body 0 is not less than twice the average, and so forth. The following piece of code shows the way to determine the first and the last element to be assigned to Rank i.

```
i = j = k = 0;
start[0] = 0;
ncumul = 0;
while ( i < (nproc-2)) {
  for (j=k; ncumul < (average*(i+1)); j++);
    ncumul += ncqsub[j];
  }
  last[i] = j-1;
  i++;
  start[i] = j;
  k = j;
}
last[nproc-1] = nbody-1;
```

4. Implementation

4.1 The Computing Platform

The program was to be written in standard C with MPI calls embedded for message passing. MPI having been chosen, the computing engine could have been any computer system connected by communication network including collections of heterogeneous machines. In the Department of CEE, a number of machines were installed with MPI. For the purpose of performance evaluation and the convenience of development activity, the system had to have some reasonable number of processors of the same capability as well as easy accessibility. Eventually, it was decided that the PC clusters in the department be used.

They are altogether 33 machines, 17 of them with 64 MB RAM and the rest 32 MB RAM, all installed with LINUX. Frequently, though, some machines became inaccessible reducing the number of available machines. Therefore only the first 15 or so machines rendered accurate figures, the rest showing only the trends.

4.2 MPI – A Message Passing Interface Standard

In message passing model of parallel algorithm, a set of uniquely identifiable processes communicate one another by sending and receiving messages. A group of processors with some local memory is currently the most widely available computing architecture. Most of message passing systems implement Single Program Multiple Data stream model of programming.

MPI is a standard for writing message passing programs. The main reason for the choice of MPI for the project was the portability and the availability. The code developed on a cluster of PCs would, with few minor modifications if any, run on a parallel supercomputer like Cray T3D. There are a number of implementations available in the public domain, for example, MPICH, LAM, CHIMP, UNIFY and WinMPI. For the current work, MPICH version is used.

Although MPI library include over one hundred functions, the basic MPI_Send and MPI_Recv together with MPI_Init, MPI_Finalize, MPI_Comm_rank and MPI_Comm_size can construct complete message passing programs. It is noted, however, that the collective communications such as MPI_Allgather and MPI_Allgatherv take time of $O(N \cdot \log N)$ in contrast to $O(N^2)$ which is inevitable when each communication is dealt with individually. Collective communication is defined as communication that involves a group of processes, and is executed by having all processes in the group call the communication routine with matching arguments. One thing to be aware of in their use is that MPI collective operations do not guarantee that all the processes participate before the host exits. Therefore, synchronisation must be provided by the implementor.

The rest of this section is a precis from MPI Standard document.

`MPI_Gather` takes eight arguments. They are `sendbuf`, `sendcount`, `sendtype`, `recvbuf`, `recvcount`, `recvtype`, `root` and `comm`. Among them `recvbuf` is the only output of the function call. Each process sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order. The outcome is as if each of the n processes in the group had executed a call to `MPI_Send(sendbuf, sendcount, sendtype, root, ...)` and the root had executed n calls to `MPI_Recv(recvbuf+i*recvcount*extent(recvtype), recvcount,recvtype,i,...)`. The `recvbuf` is ignored for all non-root processes. All arguments to the function are significant on process `root`, while on other processes, only the arguments `sendbuf`, `sendcount`, `sendtype`, `root` and `comm` are significant. The arguments `root` and `comm` must have identical valued on all processes. The specification of counts and types should not cause any location on the root to be written more than once. Note that the `recvcount` argument at the root indicates the number of items it receives from each process, not the total number of items it receives.

`MPI_Allgather` takes the same arguments except `root`, since each process take turns to be the root. The function can be thought of as `MPI_Gather`, but where all processes receive the result, instead of just the root. The j^{th} block of data sent from each process is received by every process and placed in the j^{th} block of the buffer `recvbuf`. The type signature associated with `sendcount` and `sendtype` at a process must be equal to the type signature associated with `recvcount` and `recvtype` at any other process. The outcome of a call to `MPI_Allgather(...)` is as if all processes executed n calls to `MPI_Gather(...)` for `root=0,...,n-1`.

`MPI_Allgatherv` has an extra argument which is an integer array specifying the displacement relative to `recvbuf`. Entry i specifies the displacement at which to place the incoming data from process i . Note that the `recvcount` is now an integer array containing the number of elements that are received from each process.

5. Results and Discussion

The program verification was done by running many sets of data and by tracing the loci of eleven bodies selected at regular intervals of indices. To do the simple 3-D point plots, the MATLAB package was used. The body locations after each time step were recorded into a file for all the time steps during the whole time period, and then this file was read in by the MATLAB software to produce the graphical representation of the results. A number of plotting files which contain series of the MATLAB commands were written to produce the plots. Fig.5.1 shows an example of such plots. The crosses show the initial location of the bodies, and the subsequent points issuing from them show the locations at the subsequent time-steps. The coordinates show the relative distance they travelled in three dimensional space. All the five sets of codes produced identical plots, which was taken as proof that all four sets of codes newly developed were correct, assuming that Barnes's original codes were correct.

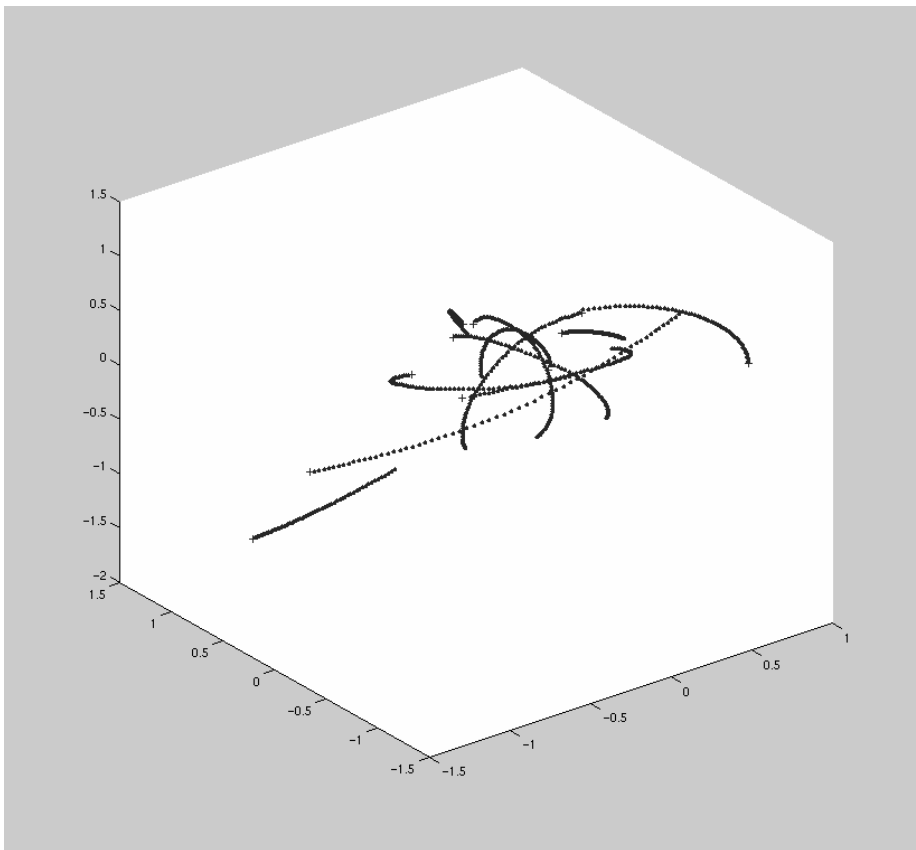


Fig.5.1 Evolution of the System

The first test carried out was to compare the execution times of Barnes's original code and the direct body-to-body code adapted in the same frame work as Barnes's code for various numbers of bodies. The "code" is Barnes's code and "n2code" is the other.

On the computing platform used, which is a cluster of PCs installed with 64 MB memory and LINUX OS, Barnes's original code could handle up to 2200 bodies and the direct body-to-body code could handle up to 3000 bodies within a couple of hours' elapsed time.

The results indicated that the direct body-to-body method, although much simpler to program, will soon become too expensive to run as the number of bodies becomes bigger.

Fig.5.2 also indicates that the overhead to construct the tree is not very great, since Barnes's code, except the very first case of 200 bodies, never took more time to execute than the other code. It was expected that Barnes code would outperform the other when many bodies were dealt with. However, it was thought possible for the direct body-to-body code to perform better with small numbers of bodies to deal with, since the tree building is an extra overhead in a hierarchically structured algorithm like Barnes-Hut's.

Bodies	n2code	code
200	1.65	1.69
400	7.78	5.07
600	17.53	9.30
800	31.11	14.17
1000	48.65	19.17
1024	51.91	20.41
1200	70.00	24.91
1400	85.67	30.85
1600	111.87	36.81
1800	141.59	33.23
2000	174.80	49.99
2200	210.56	56.49
2400	251.79	
2600	295.46	
2800	342.82	
3000	403.14	

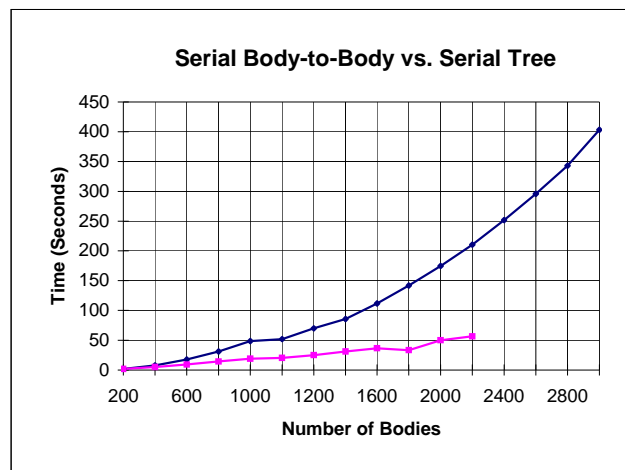


Fig.5.2 Comparison between N^2 Algorithm and $N \cdot \log N$ Algorithm

Bodies	llv	llu	code
200	1.87	1.86	1.69
400	5.28	5.17	5.07
600	9.62	9.41	9.30
800	14.40	14.21	14.17
1000	19.40	19.37	19.17
1200	25.46	24.94	24.91
1400	30.60	30.63	30.85
1600	36.94	36.89	36.81
1800	42.89	43.02	43.23
2000	49.63	49.59	49.99
2200	55.98	56.27	56.49

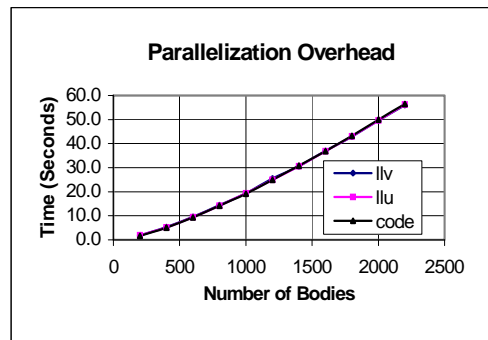


Fig.5.3 Serial Version and Parallel Version of Barnes's Code

Fig.5.3 shows the times of Barnes's code and its two parallel versions on one processor. The difference between them is the overhead for parallelisation. It is noticed that the overhead is marginal. The curves are overlapping one on top of the others. Only the table on the left shows the small differences. The same is true with the body-to-body code and its parallel version as shown in Fig.5.4.

Bodies	n2code	n2codell
200	1.65	1.80
400	7.78	7.32
600	17.53	15.84
800	31.11	27.95
1000	48.65	43.45
1200	70.00	62.42
1400	85.67	87.47
1600	111.87	113.96
1800	141.59	144.35
2000	174.80	178.01
2200	210.56	215.44
2400	251.79	256.23
2600	295.46	302.29
2800	342.82	348.77
3000	403.14	411.60

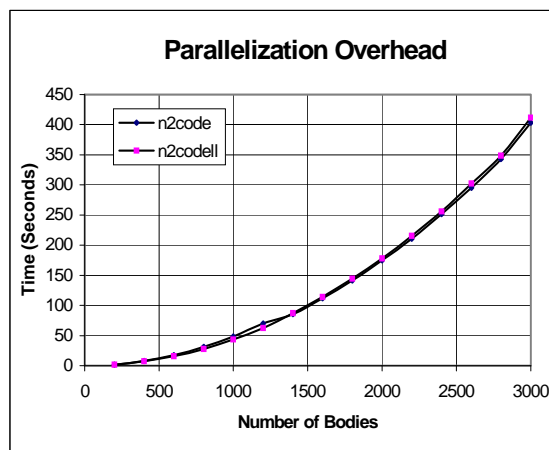


Fig.5.4 Serial Version and Parallel Version of Body-to-Body Code

Fig.5.5 below shows the execution times of the parallel codes to deal with 4096 bodies. "llu" is the parallel version of the Barnes's code with uniform number of bodies for each processing element. "llv" is the parallel version of the Barnes's code with dynamic load balancing. The curves of "llu" and "llv" are virtually overlapping each other. The "n2codell" is the parallel version of the body-to-body code.

Using three processors the direct body-to-body code took about 10 times more execution time to deal with 8192 bodies. With one processor it took about 12.5 times, and with more than three the ratio becomes smaller and smaller as more processors are employed. This implies that the direct body-to-body code has a much better scalability characteristic than the parallel version of Barnes's code.

As discussed in Section 2.5, the N-body problem is a typical example which intrinsically has high parallelism. Thus it is well known that body-to-body codes vectorise well. [Hockney (1988)] However, this does not discard the need for the more efficient hierarchical algorithm when many processors are employed. The difference between $O(N^2)$ and $O(N*\log N)$ remains however many processors are used for the calculation. It is a matter of how efficiently the code can be parallelised, in other words, how suitable the algorithm is for parallelisation. [Moldovan (1993)]

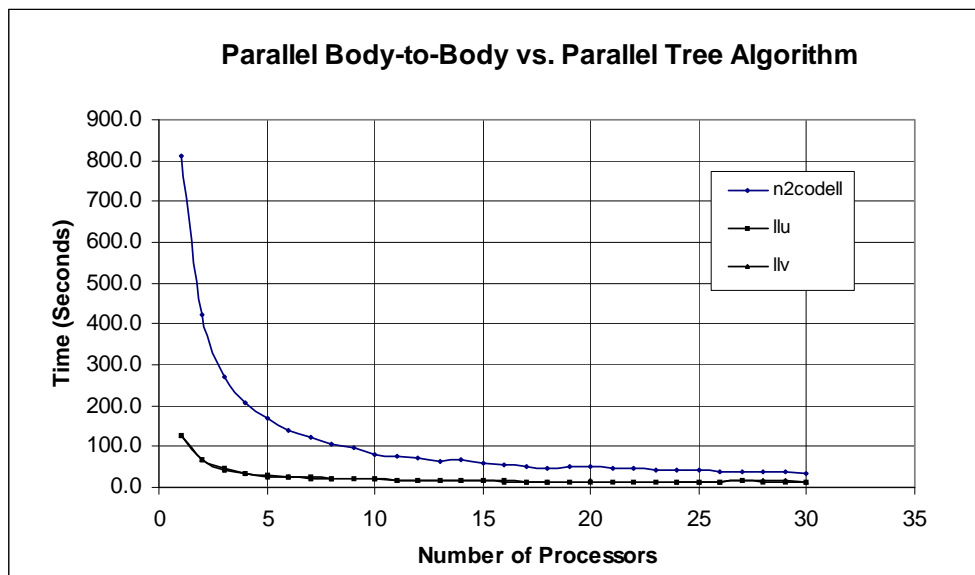


Fig.5.5 Time Comparison of N^2 Algorithm and $N*\log N$ Algorithm

Fig.5.6 shows the speedups of the parallel codes. As discussed above, the body-to-body code shows an excellent speedup characteristic – almost linear. On the other hand the parallel version of Barnes's code shows relatively poor speedups. That was in fact not because the code was not readily parallelisable, but because the part of the code for tree construction had not been parallelised. Due to the limited time for the project, the task was left for future work.

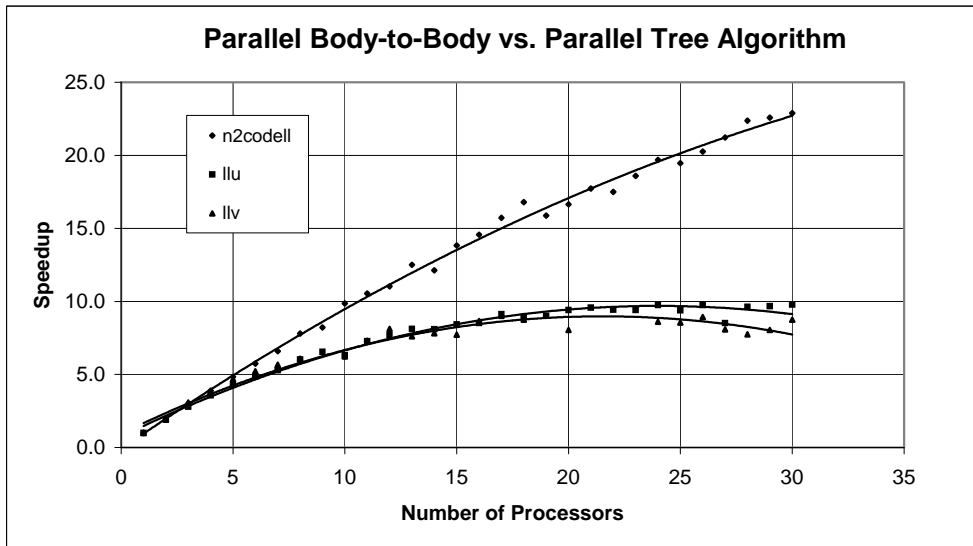


Fig.5.6 Speedups of N^2 and $N \cdot \log N$

The justification for not including the task was that the tree construction cost only less than 5 % for the serial code. This was mentioned by Barnes himself and confirmed by the profiling utilities available on unix, namely “prof” and “gprof”. However, as the number of processing elements increases, the computation load for each processing element diminishes, the total amount of work being shared among the processing nodes participating in the computation, whereas the task of full tree construction was still on every processing element. Thus by the time 20 processors are involved, almost half of the execution time of each processing element was spent on tree construction. Therefore, the poor speedups of the parallel tree-code was predicted before the actual tests were carried out. Nevertheless, the speedups shown for a small number of processors clearly indicates the potential of producing good speedup characteristics, once the tree construction part is likewise parallelised.

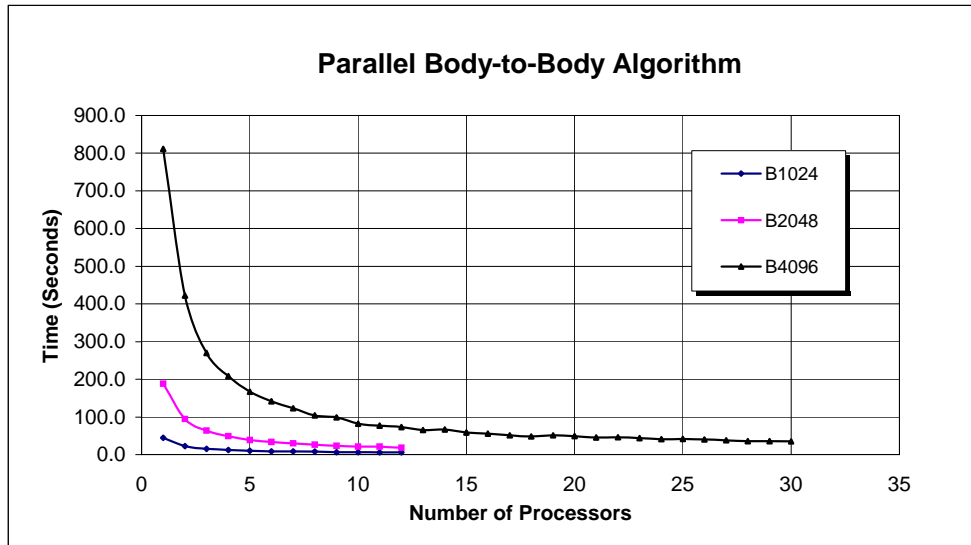


Fig.5.7 Times taken by the Parallel Body-to-Body Code

Fig.5.7 shows the times taken by the parallel body-to-body code. The times are reducing as the number of processors increases. Although they are all changing in a consistent manner, it is noticeable that the more bodies are dealt with, the steeper the changes are.

Figure 5.7 to 5.12 show the performance of each parallel code. They show the times and speedups of each code varying the problem sizes and the platform sizes. Overall, they scale up well and perform better with coarser granularity of problem.

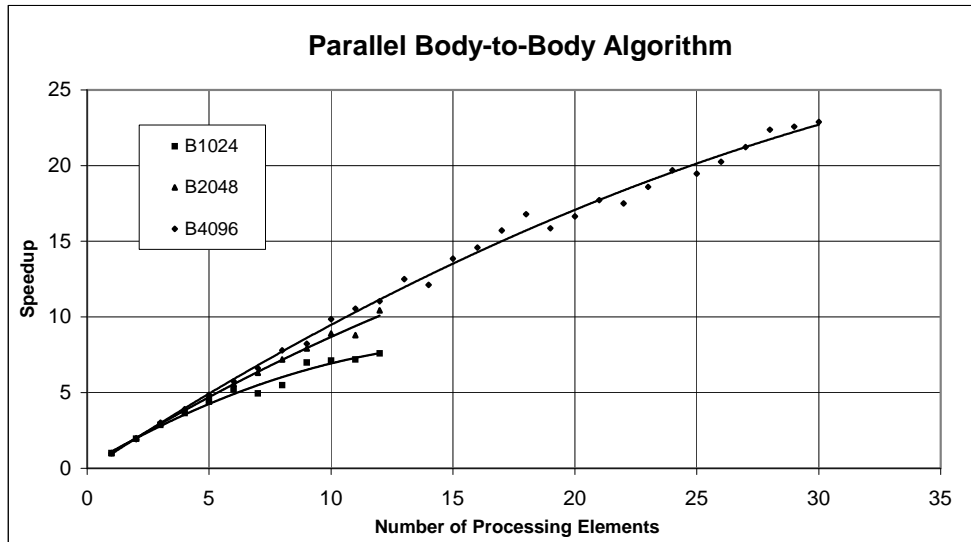


Fig.5.8 Speedup of Parallel body-to-Body Code

The speedup curves shown in Fig.5.8 clearly demonstrate the excellent speedup characteristic of the direct body-to-body algorithm. They are almost linear and also indicate that bigger problems scale better.

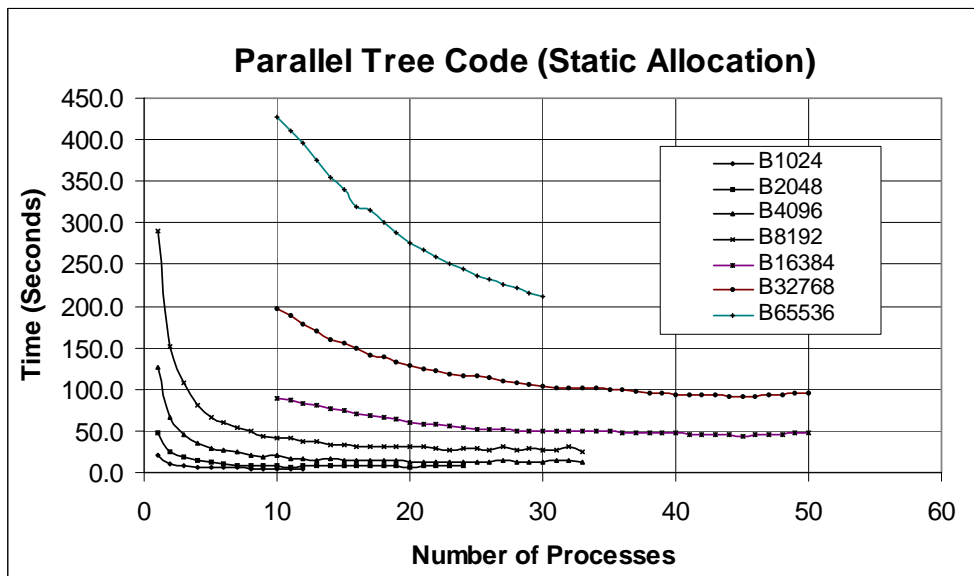


Fig.5.9 Times of Parallel Tree Code with Static Data Allocation

For the whole range of problem sizes, the computing times consistently diminished with increased numbers of processors.

Two interesting points were observed. The actual numbers of processors were 16 for the 64MB RAM machines and 15 for the 32MB RAM machines. There was no noticeable change in curvature after the first 16 processors which were 64MB RAM machines and followed by the 32MB RAM machines, implying that the granularity of computation was small enough to be handled by the smaller machines.

With more than 31 processors, the times remained more or less constant, even though some of the machines were managing two processors. This may be implying that the communication overhead far outweighs the computation load with that many processors involved. Otherwise, it was thought, the computing time would increase almost twice as much, because the processors with two processes would take twice as much time to complete their tasks before each synchronisation.

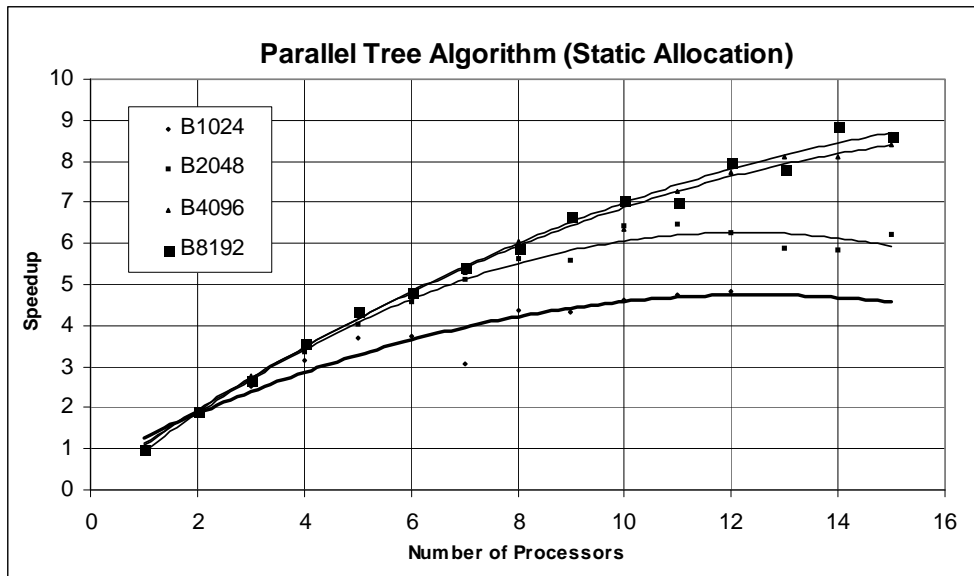


Fig.5.10 Speedup of Parallel Tree Code with Static Data Allocation

The curves show much poorer speedup characteristic than that of the direct body-to-body method. With smaller numbers of bodies, say 1024 or 2048, the speedups were already reducing when more than 10 processors were employed.

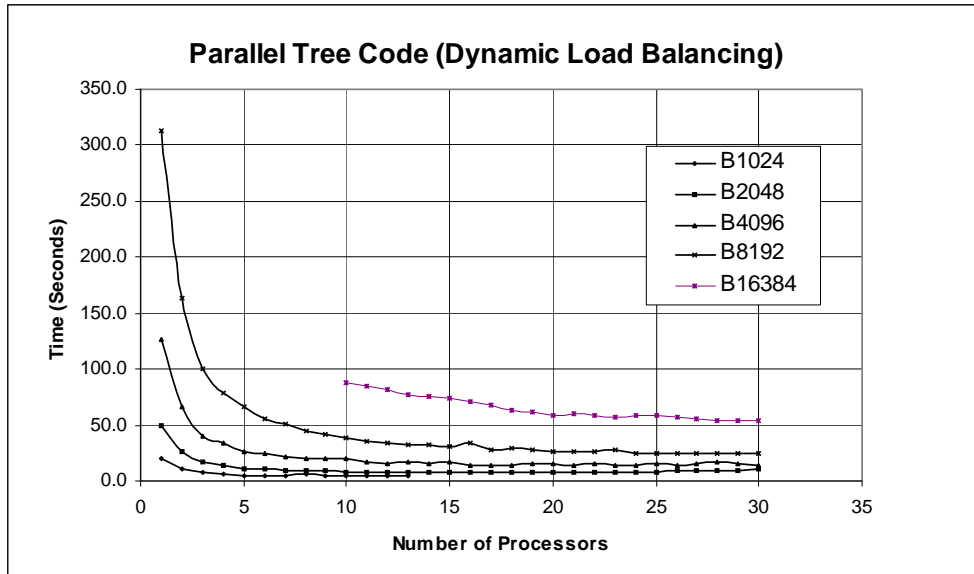


Fig.5.11 Parallel Tree Code with Dynamic Load Balancing

Fig.5.11 again shows consistently diminishing curves. Overall, the curves were very much similar to those of static allocation.

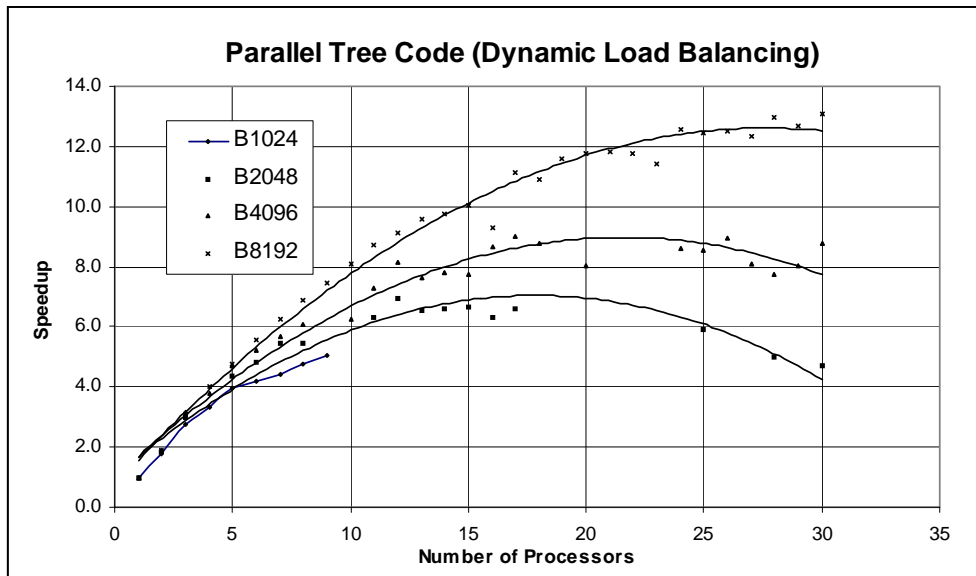


Fig.5.12 Speedups of Parallel Tree Code with Dynamic Load Balancing

Here it is again clearly shown that a serial part in the code is a serious hindrance in performance. In this case, the overhead in distributing the load dynamically seems also to be the culprit. Although it is a small piece of the calculation, it has not been parallelised and seriously affects the parallel performance together with the tree building part of the code.

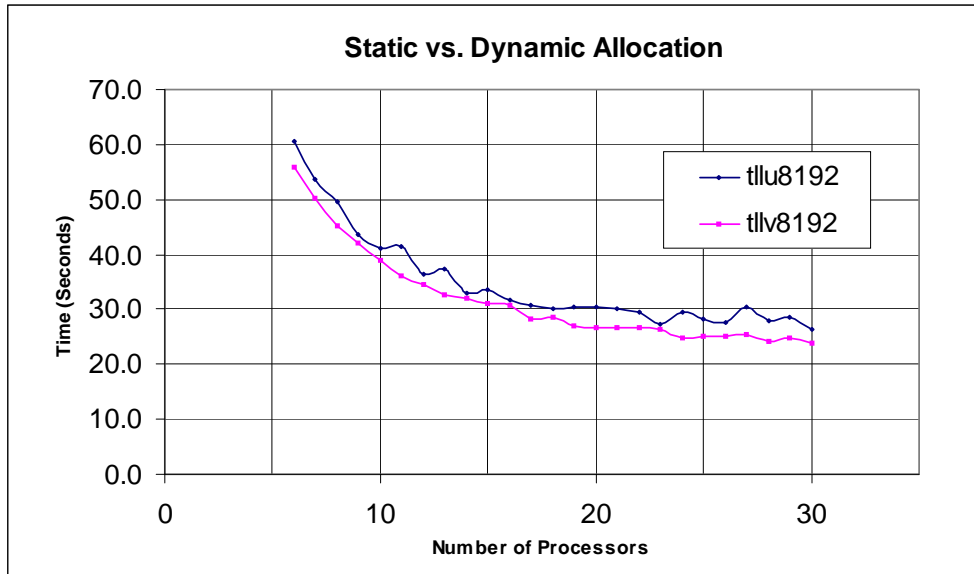


Fig.5.13 Times for Parallel Codes with Static and Dynamic Load Allocation

Fig.5.13 shows the time gains of the dynamic load balancing over the static data allocation. An average of 5.48% gains are shown in the data range. When a bigger problem is dealt with and allowed to evolve a longer period of time, even bigger gains are expected. This is because the formation of clusters is the very factor necessitating the dynamic load balancing. Therefore, when clusters have formed the effects of the dynamic load balancing will be more pronounced.

6. Conclusion

A serial code of the direct body-to-body algorithm has very limited applicability due to the prohibitively large computing time that increases in proportion to the square of the number of bodies. The hierarchical tree code, in contrast, taking time proportional to $N \cdot \log N$ can handle much bigger problems.

Barnes's code of hierarchical tree structure incurs only small overheads for the tree construction and proved to be very efficient to deal with N -body problems. However, it also reaches its limit far too early to deal with realistic problems.

Parallel processing seems to be the only way to manage real problems at the moment. N -body problems are ideally suited to parallelisation, and with an almost trivial effort it can be done. The parallel version of body-to-body code showed excellent scalability. The parallel versions of Barnes's tree-code demonstrated the potential to show the same excellent speedup characteristics, if only the tree construction part had been parallelised too.

To exploit fully the benefit of the hierarchical tree algorithm, a form of dynamic load balancing is required. Otherwise, the optimum performance of the computing platform can not be achieved. A simple schedule of dynamic load balancing could demonstrate some gains in execution time, even without the parallelisation of the process itself.

The MPICH implementation of MPI is a very efficient message passing tool and very easy to use. Particularly, the collective operations such as `MPI_Allgather` and `MPI_Allgatherv` which were used in the main part seems to be very efficient – involving operations of $O(N \cdot \log N)$ rather than $O(N^2)$ - and simple to use.

The ultimate aim of this project was developing a parallel hierarchical tree-code with dynamic load balancing for the calculation of N -body problems. The code developed is incomplete in that the tree construction part and dynamic load allocation part of the code have not been parallelised which causes the performance of the code to be seriously impaired when many processors are engaged in the calculation. Parallelising the load allocation part is straightforward, but parallelising tree construction is by no means simple. [Goil (1995)] There is no limit to the number of bodies to be used as far as the software is concerned, but of course the hardware does impose such a limitation. No effort has been made to eliminate the limitation imposed by the memory size. The computing platform can be anything including a collection of heterogeneous computers, as long as an MPI implementation is installed.

Further improvement in performance could be made if a more efficient way of dynamically allocating the load can be found. When augmented N -body problems are dealt with, the present method of dynamic load balancing could become less effective. This is because the distribution of bodies could change abruptly after each time-step which is a direct violation of the assumption made for the justification of using the information of the previous time-step for the calculation of the following time-step.

When a great many processors are used to deal with a large number of bodies, the simple body-to-body method with the excellent scalability and little overhead may outperform the more sophisticated hierarchical tree algorithm unless this is really well parallelised.

References

1. Austin, W. J. and Wallace, A. M., Object Location by Parallel Pose Clustering, Heriot-Watt University (1997)
2. Barnes, J.E., N-Body models of Collisionless Systems
3. Becciani, U., A parallel tree code for large N-body simulation: dynamic load balance and data distribution on a CRAY T3D system, Computer Physics Communications 106 (1997) 105-113
4. Bergmann, Peter G, The Riddle of Gravitation, John Murray (Publishers) Ltd (1969)
5. <ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode>
6. Goil S. and Ranka, S., Parallelization Requirements for Hierarchically Structured Applications on Coarse-Grained Parallel computers
7. Goil S. and Ranka, S., Parallelization of Hierarchically Structured Applications on Distributed Memory Machines, NPAC Technical Report SCCS-688 (1995)
8. Grama etc, Scalable Parallel formulations of the Barnes-Hut method for n-body simulations, Parallel Computing 24 (1998) 797-822
9. Grama, A.Y., N-Body Simulations Using Message Passing Parallel Computers, SIAM Conference on Parallel Processing, San Francisco (1994)
10. Hockney, R.W. and Jesshope, C.R., Parallel Computers 2 (1988) ISBN 0-85274-812-4
11. <http://www.icsi.berkeley.edu/cs267/lecture26/lecture26.html>
12. <http://www.infomall.org/npac/pcw/node278.html>
13. <http://www-users.cs.umn.edu/~kumar>
14. Karypis, G. and Kumar, V., Effective Parallel Formulations for Some Dynamic Programming Algorithms, University of Minnesota (1992)
15. Kumar, V. etc, Scalable Load Balancing Techniques for Parallel Computers
16. Lamping, J. and Rao, R., Visualising Large Tress Using the Hyperbolic Browser, Xerox Palo Alto Research Centre
17. Moldovan, Dan I, Parallel Processing from Applications to Systems (1993) ISBN I-55860-254-2
18. MPI: A Message-Passing Interface Standard, May 5, 1994
19. Pereira, N.S.A., A Parallel N-Body Integrator using MPI, Lecture Notes in Computer Science Vol.1573 (1999) 627-639
20. Pringle, G.J., Numerical study of three-dimensional flow using fast parallel particle algorithms, Maths Dept., Napier University (1994)
21. Wallace, A.M. etc, A Dual Source, Parallel Architecture for Computer Vision, The journal of Supercomputing, 12 (1998) 37-56