Report Number: WUCS-93-16

1993-01-01

The N-body Problem: Distributed System Load Balancing and Performance Evaluation

Vasudha Govindan and Mark A. Franklin

In this paper, the N-body simulation problem is considered, its parallel implementation described, its execution time performance is modeled and compared with measured results, and two alternative load balancing algorithms for enhancing performance investigated. Parallel N-body techniques are widely applied in various fields and possess characteristics that challenge the computation and communication capabilities of parallel computing systems and are therefore good candidates for use as parallel benchmarks. Performance models may be used to estimate the performance of an algorithm on a given system, identify performance bottlenecks and study the performance implications of several algorithm are system enhancements. In this... **Read complete abstract on page 2.**

The N-body Problem: Distributed System Load Balancing and Performance Evaluation

Vasudha Govindan and Mark A. Franklin

Complete Abstract:

In this paper, the N-body simulation problem is considered, its parallel implementation described, its execution time performance is modeled and compared with measured results, and two alternative load balancing algorithms for enhancing performance investigated. Parallel N-body techniques are widely applied in various fields and possess characteristics that challenge the computation and communication capabilities of parallel computing systems and are therefore good candidates for use as parallel benchmarks. Performance models may be used to estimate the performance of an algorithm on a given system, identify performance bottlenecks and study the performance implications of several algorithm are system enhancements. In this paper, we propose a general framework for developing performance models for a class of synchronous iterative algorithms and specilize it for the case of N-body simulation algorithms. We use the model to estimate the performance of two load balancing algorithms.

The N-body Problem: Distributed System Load Balancing and Performance Evaluation

Vasudha Govindan and Mark A. Franklin

WUCS-93-16

March 1993

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130-4899

Abstract: In this paper, the N-body simulation problem is considered, its parallel implementation described, its execution time performance is modeled and compared with measured results, and two alternative load balancing algorithms for enhancing performance investigated. Parallel N-body techniques are widely applied in various fields and possess characteristics that challenge the computation and communication capabilities of parallel computing systems and are therefore good candidates for use as parallel benchmarks.

Performance models may be used to estimate the performance of an algorithm on a given system, identify performance bottlenecks and study the performance implications of several algorithm and system enhancements. In this paper, we propose a general framework for developing performance models for a class of *synchronous iterative algorithms* and specialize it for the case of N-body simulation algorithms. We use the model to estimate the performance of two load balancing algorithms.

The N-Body Problem: Distributed System Load Balancing And Performance Evaluation*

Mark Franklin
jbf@random.wustl.edu

Vasudha Govindan
vasu@wuccrc.wustl.edu

Computer and Communications Research Center
Washington University
Campus Box 1115
One Brookings Drive
St. Louis, Missouri 63130

1 Introduction

In this paper, the N-body simulation problem is considered, its parallel implementation is described, its execution time performance is modeled and compared with measured results, and two alternative load balancing algorithms for enhancing performance are investigated.

Parallel N-body techniques are widely applied in a number of fields[8] ranging from astrophysics[11], to fluid dynamics, to computational geometry[7]. They require dynamically changing, non-uniform, intensive computation and irregular, unstructured communication. They are therefore good candidates for use as parallel computing benchmarks[10] and the results presented in this paper are part of an ongoing effort at developing such parallel benchmarks. In addition, the N-body simulation algorithm presented here is prototypical of a wide class of algorithms referred to as *synchronous iterative algorithms.* The models and results given in this paper apply to much of this class. N-body simulation algorithm has been implemented on a network of SUN workstations connected by standard ethernet, running under the PVM[13] environment. Performance has been measured and models verified in this manner.

An important step in the design of high performance computing systems is to study possible enhancements to the existing system and estimate the performance implications of these enhancements. Similarly, certain modifications to the algorithms may also lead to significant performance improvements. To evaluate the these modifications and their performance implications, a general performance model is needed that can predict the performance of an algorithm on a particular system. This paper presents a general framework for developing simple mean-value oriented performance models for a class of synchronous iterative algorithms. The performance model is then specialized for the case of the parallel N-body

*This research has been sponsored in part by funding from the NSF under grant CCR-9021041.

simulations algorithm. Using this model, we estimate the performance improvements offered by alternative load balancing techniques. The model can also be used to examine the effects of other algorithm enhancements (e.g., latency hiding techniques), and also to determine the effects of architecture parameters on performance (e.g., the use of hardware based synchronization networks). These investigations, however, are not pursued here.

The remainder of this paper is divided into five sections. Section 2 describes the N-body simulation problem and the serial and parallel algorithm used in our implementation. Section 3 states the various steps involved in the development of the performance model for the N-body simulation algorithm. The model can be used to predict the performance of the algorithm on a given system and in Section 4 it is used to estimate algorithm performance when executing on a network of SUN workstations. Measured performance is compared with model predicted results and the two are shown to be in good agreement. In section 5, we evaluate the effect of two load balancing algorithms on the performance of the simulation. We show that the distributed load balancing algorithm yields up to 25% improvement in speedup. Section 6 summarizes our work, presents conclusions and indicates how the model can be used to explore other potential performance improvements.

2 N-Body Simulations

Classical N-body techniques study the evolution of a system of bodies under forces exerted on each body by the entire ensemble. For n bodies, a direct evaluation of all pairwise forces results in a time complexity of $O(n^2)$. Approximate methods like the hierarchical methods considered here reduce the complexity to $O(n \log n)$ for general distributions and $O(n)$ for uniform distributions[2][1].

Hierarchical methods are based on the following principle due to Sir Isaac Newton[4]: If the magnitude of interaction between bodies falls off rapidly with distance, then the effect of a large group of bodies may be approximated by a single equivalent body, if the group of bodies is far enough away from the point at which the effect is being evaluated.

The technique discussed here is based on the Barnes-Hut algorithm[2] to study the evolution of a system of bodies (or particles) under the influence of Newtonian gravitational attraction. The input consists of the mass, initial position and initial velocities of particles distributed over a finite physical domain. The simulation proceeds over a number of time-steps, every time-step computing the net force on every body and updating its position and other attributes. The simulations enable us to study the evolution of such a system over time. A two dimensional physical domain is considered here, though the technique can be

```
ComputeForce (cell, particle)
begin
if ((length (cell) / distance (cell, particle) ≤ θ) or (cell is a leaf))
    force = force + NewtonForce (cell, particle);
else
    For each subcell of cell, Do:
        ComputeForce (subcell, particle);
end.
```

Figure 1: **The Recursive Force Calculation Algorithm**

easily extended to the three dimensional case.

2.1 The Algorithm

As indicated earlier, the Barnes-Hut algorithm uses a hierarchical approach and results in a computational complexity $O(n \log n)$. To do this, all the information about the particles are represented in the form of a binary tree.

The binary tree is formed using a procedure called *Orthogonal Recursive Bisection (ORB)*[6]. The procedure starts at the root of the binary tree which represents the entire physical domain and all particles contained in it. Successive levels of the tree are formed by recursively dividing the physical domain into two rectangular sub-domains (with equal number of particles) and assigning tree-nodes to represent each of these sub-domains until the final sub-domains assigned contain just one particle each. In the resultant tree structure, leaves represent particles and internal nodes represent groups of particles. While the original Barnes-Hut algorithm employs quad-trees, our implementation employs binary trees.

The next step associates with each node, the center of mass and the equivalent mass for particles in that node's sub-region. This is done by propagating information up the tree from the leaves to successive "parent" nodes, and finally to the root. The leaves now contain information about the particles, and the internal nodes contain information about groups of particles. The tree can now be used to compute the resultant force on each particle. The tree is traversed once per particle to compute the resultant force acting on that particle by all other particles (or particle aggregates). The force computation algorithm for each particle starts at the root of the tree and proceeds downward. At each internal node, a calculation is made to determine whether the aggregate of particles represented by the node can be considered as a single mass. This is done by calculating the ratio of the length of the sub-domain currently being processed, length(node), to the distance between the particle

```
Nbody_Simulation()
Begin
Initialize mass, position and velocities of particles.
For each timestep, Do:
    Begin
    Build binary tree.
    Compute total mass and center of mass for each cell.
    For each particle, Do:
        Begin
        ComputeForce(root,particle);
        Update position, velocities.
        End;
    End;
End.
```

Figure 2: Serial N-Body Simulation Algorithm

and the center of mass of the sub-domain, **distance**(node, particle). If the sub-domain is far enough away from the particle (*i.e.,* if **length**(node) / **distance**(node, particle) $\leq \theta$, where θ is a fixed accuracy parameter), the Newtonian force between the particle and the sub-domain is computed. Otherwise, the algorithm is recursively applied to each child of the current node (see Figure 1).

At the end of the force calculation phase, the resultant positions and velocity of the particles are updated. Given that the particles now have new positions, a new tree is formed and the entire procedure is repeated for the next timestep. The simulation proceeds for the desired number of timesteps as summarized in Figure 2.

2.2 Parallel Implementation

Two important issues in the parallel implementation of the algorithm described above are:

- Partitioning of the problem into modules, and the effective mapping of these modules onto the available processors (*i.e.,* to obtain maximum speedup): For the N-body problem, the entire physical domain is broken into sub-domains and the mapping assigns each processor to one sub-domain. Processors are responsible for all computation associated with the particles within their sub-domain.

- Imposing a structure on the communication pattern required by the algorithm such that the number and size of messages are minimized: In the N-body problem, the computation of resultant forces requires that the mass and position of every other

```
Build_Locally_Essential_tree()
Begin
for each of the log2 p bisectors
     Begin
     Traverse the tree and identify data
          that may be essential on the
          other side of the bisector
     Delete data that can never be necessary on
          this side of the bisector.
     Exchange data with corresponding processor
          on the other side of the bisector.
     Merge received data into tree
     End
End
```

Figure 3: Algorithm for "Locally Essential Tree" formation

particles (or particle ensemble) be known. When particles are distributed over all processors, exchange of information is required between every pair of processors resulting in long-range, unstructured communications.

A decomposition of the physical domain, using the *Orthogonal Recursive Bisection (ORB)* described earlier, ensures rough computational balance while providing a structure for efficient communication. The physical domain is recursively divided into two rectangular domains (with equal workloads) and half the processors are assigned to each domain. The procedure is repeated until there is one processor associated with each rectangular domain. To obtain regularly shaped partitions, the cartesian direction in which division takes place is alternated with successive divisions. At the end of the partitioning phase, the physical domain is decomposed into p sub-domains, where p is the number of available processors. The ORB decomposition results in $\log_2 p$ spatial bisectors, one for each recursive bisection. The decomposition procedure is represented in the form of a binary tree, called the *ORB-tree* where, each level of the tree corresponds to a spatial bisector, and the leaves of the tree represent the processor sub-domains.

Each processor now has a spatially contiguous domain and is responsible for all computation associated with the particles present in its domain. The processors now recursively bisect their respective domains and represent their particles in the form of a binary tree. The information at the intermediate nodes, the center of mass and equivalent mass of the sub-domains are filled by an upward pass from the leaves to the root of the local tree.

```
Parallel_N-Body_Simulation()
Begin
Initialize mass, position and velocities of particles
Distribute particles among processors
For Each Time-Step,
 At Each Processor, Do
   Begin
   Form Complete ORB tree
   Compute Center of mass of local particles
   Exchange Center of mass (Synchronization Phase)
   Build_Locally_Essential_Tree()
   For Each Local Particle, Do:
       Begin
       ComputeForce(root, particle)
       Update Position, Velocity
       End;
   Invoke Load Balancing Routines
   End;
End.
```

Figure 4: The Parallel N-Body Simulation Algorithm

To compute the center of mass and the equivalent mass of the entire physical domain, the processors need to exchange information on the center of mass and equivalent mass of their respective domains. Information is passed from the leaves (in this case, the processor domains) to the root, where the center of mass and equivalent mass of the entire domain is calculated and then, passed back down the tree so that all the processors have the information. This can be viewed as an implicit synchronization phase in the algorithm since all the processors have to "co-operate" to compute the total mass and center of mass of the system.

The *ORB-tree* associated with particles local to the processor and the relevant information on particles from other processors, together, is referred to as the "locally essential tree"(*le-tree*). Clearly, formation of the *le-tree* involves exchange of information between each pair of processors. The ORB decomposition imposes a hierarchical structure on this apparently "unstructured" communications requirement. The algorithm to construct the *le-tree* (due to Salmon [11]) is illustrated in Figure 3. The *le-tree* algorithm reduces the amount of data transferred (and hence the message size) by having the sender identify and send only the data required by the receiving processors. Due to the tree structure, the exchange of data takes place in stages corresponding to each level of the tree. Successive messages aggregate the data received, thus limiting the total number of messages exchanged. Each processor

sends and receives $\log_2 p$ messages (as opposed to p messages in the unstructured case). In addition to reducing the number of messages sent, the cost of the messages is reduced if a binary tree structure can be effectively mapped onto the underlying communication network topology. This is true for many of the topologies available (e.g., hypercube).

Once the *le-tree* is formed, the processors have all the information needed to compute the resultant force on the particles in their domain. Resultant force computation takes place entirely locally without any communication using a recursive function similar to the algorithm in Figure 1. The resultant velocity and displacement of the particles are updated. Figure 4 summarizes the parallel N-body simulation algorithm in pseudo-code form. In the following section, a performance model is developed to evaluate the performance of the parallel N-body simulation algorithm on any given system.

3 Performance Model

3.1 Synchronous Iterative Algorithms

Performance models may be used to estimate the performance of an algorithm on a particular computer system and also to evaluate potential enhancements to the system and to the algorithm. In this section, we propose a general performance model that can be adapted to derive performance models for a class of synchronous iterative algorithms (also called multiphase algorithms or synchronous algorithms)[9][5]. The model formulation is similar to the technique developed in [3] for hierarchical discrete event simulation.

Synchronous iterative algorithms generally possess the following structure: The execution of the algorithm proceeds in a number of sequential steps or "iterations". The work associated with each iteration is distributed over some or all of the available processors. At the end of each iteration, the processors perform a *barrier synchronization*[12] and then proceed to the next iteration.

The time spent in each iteration can be divided into three phases - computation, communication and synchronization. The work done in each of these phases depends on the algorithm. Typically, in the computation phase, each processor evaluates a set of local variables or attributes using its own local unshared data. Local data here refers to data present in the processor memory for distributed memory MIMD systems, and in a processor's local cache for shared memory MIMD systems.

In the communication phase, each processor (in a distributed memory architecture) sends some or all of its local information to the processors that may need the information to compute their local variables and receives similar information from other processors. In

this development, we assume a distributed memory system although it applies with minor modifications to the shared memory case.

The synchronization phase may be just a barrier point that ensures that all processors complete iteration $k-1$ before any processor can start iteration k. In distributed memory systems, synchronization involves exchange of messages between processors ($\log_2 p$ messages for a complete exchange on hypercube-type systems).

The synchronization phase forces the processors to proceed in lock-step, one iteration at a time, through the algorithm. Therefore the time for each iteration is the sum of the synchronization time and the maximum, over all processors, of the sum of computation and communication times on each processor. Here, we assume that computation and communications are not overlapped. However, the model can be extended to allow partial or complete overlap. The time for each iteration, T_{iter}, given p processors are available can be expressed as:

$$T_{iter} = \max_{i=0}^{p} (T_{comp,i} + T_{comm,i}) + T_{sync} \quad (1)$$

In the ideal case, where the computation and communication loads are evenly distributed over all processors, the terms in the Max expression reduce to their average values.

Smaller T_{iter} results in smaller execution times and hence better performance. A number of algorithm and system enhancements can be applied to improve performance. Some options are listed below:

- High performance communication support: Reduces T_{comm}.
- Special hardware/software for synchronization: Reduces T_{sync}.
- Ideal Load balancing: Reduces $Max(.)$ term to its average value.
- Overlapping computation and communication: Reduces the $(T_{comm,i} + T_{comp,i})$ term to a max $(T_{comm,i}, T_{comp,i})$ term.

In the remainder of this section, we develop expressions for $T_{comp,i}$, $T_{comm,i}$ and T_{sync} for the N-body simulation algorithm. Later we focus on improving performance by achieving a better load balancing.

3.2 Performance model for N-body Simulation algorithm

In developing the model for the N-body simulation algorithm, we assume a distributed memory parallel processing system having identical processors and communicating over an interconnection network. The communication costs depend on network topology and technology.

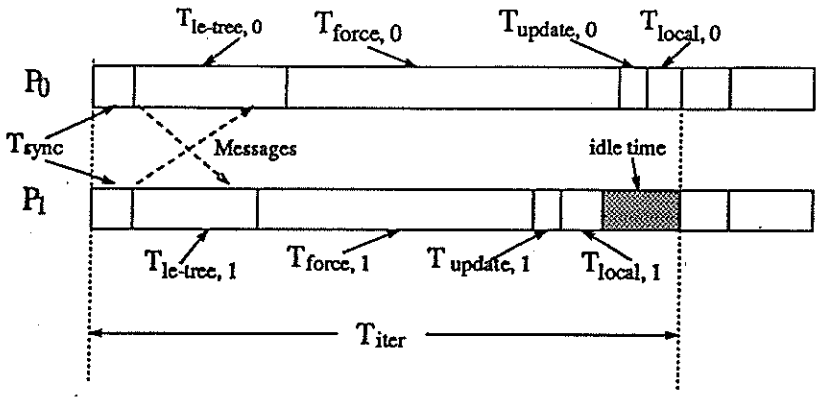

Figure 5: Sample Iteration of the Algorithm

Algorithm initialization costs are considered to be negligible. The various steps involved in each iteration can be grouped into computation, communication and synchronization phases. Each iteration in the algorithm can be divided into the following five steps.

1. Exchange/propagate local center of mass (synchronization): T_{sync}.
2. Form *le-tree* : $T_{le-tree}$ (corresponds to T_{comm}).
3. Compute resultant forces: T_{force} (part of $T_{comp,i}$).
4. Update particle attributes: T_{update} (part of $T_{comp,i}$).
5. Form new tree of local particles: T_{local} (part of $T_{comp,i}$).

Figure 5 shows a sample iteration of the algorithm for two processors. Table (i) lists the variables used in the model and their definitions. Equation 1 for T_{iter} (for p processors) can now be written as follows:

$$T_{iter} = T_{sync} + \max_{i=1}^{p} (T_{le-tree,i} + T_{force,i} + T_{update,i} + T_{local,i}) \quad (2)$$

Due to the dynamic nature of the simulation, the workload on the processors may not be balanced. Imbalance in workload causes the lightly loaded processors to wait for the heavily loaded ones to reach the barrier. However, since the N-body simulations are compute intensive, the force computation phase dominates each iteration[2]. The imbalance in all other phases are generally very small and the max function associated with $T_{local,i}$, $T_{le-tree,i}$ and $T_{update,i}$ may be approximated by their respective average values. Equation 2 can therefore be rewritten as:

$$T_{iter} = T_{sync} + T_{le-tree} + \max_{i=1}^{p} (T^i_{force}) + T_{update} + T_{local} \quad (3)$$

[2]Our experiments, discussed later, show that in a 1024-particle, 4-processor simulation, 85% of the time is spent in the force computation phase.

Table i: Variables and Definitions

N	Total number of Particles simulated.
p	Number of Processors.
n_i	Number of particles allocated to processor i.
$\bar{n}$	Average number of particles per processor. $\bar{n} = N/p$
T_{iter}	Time to complete one iteration.
T_{sync}	Synchronization time per iteration
$T_{le-tree,i}$	Time to form *le-tree* on processor i
$T_{force,i}$	Time to compute forces on processor i
$T_{update,i}$	Time to update particle information on processor i
$T_{local,i}$	Time to form local tree on processor i
$T_{msg}(s)$	Message transmission time for message of length s.

A convenient way to deal with the max term is to define an imbalance factor, β to be the ratio of the maximum time for force evaluation to the average force evaluation time.

$$\beta = \frac{\max_{i=1}^{p} (T_{force}^{i})}{T_{force}} \tag{4}$$

β may change considerably as the simulation progresses, beginning with a value of about 1 (reflecting good initial balance) and moving to values in the 2 to 3 range as the load balancing deteriorates. Equation 3 can now be rewritten as:

$$T_{iter} = T_{sync} + T_{le-tree} + \beta \times T_{force} + T_{update} + T_{local} \tag{5}$$

Detailed expressions for each of the terms in equation 5 are now derived. Table (ii) defines the model parameters and their values measured on our system (described later). Though the model can be extended to general topologies, to simplify notation and message time expressions it is assumed that a binary tree can be embedded in the communications network topology (e.g., hypercube).

3.2.1 Synchronization

In the synchronization phase, the processors exchange the total mass and center of mass of their respective domains to calculate the total mass and center of mass of the entire system. The synchronization procedure makes use of the ORB tree structure to compute and distribute the global information. Processors at the leaves of the tree structure send the information about their domain to their respective "parent" processors. The parent computes the total mass and center of mass of its "children" and passes the information to

Table ii: Model Parameters

Parameter	Definition	Measured Value	
K_{eval}	Force evaluations per particle	17.43	
N_{byte}	No. of bytes to represent a particle	80	bytes
t_{eval}	Time per force evaluation	100	μs
$t_{update-p}$	Update time per particle	70	μs
$t_{local-p}$	Local tree formation time per particle	80	μs
$t_{search-p}$	*le-tree* search time per particle	300	μs
$t_{merge-p}$	*le-tree* merge time per particle	300	μs

its parent. The "root" processor computes the global information and passes it down the tree in a similar fashion.

The synchronization procedure therefore takes $2 \times \log_2 p$ message times corresponding to the $\log_2 p$ levels of the tree. The processing at intermediate processors is negligible (few additions). Thus:

$$T_{sync} = 2 \times \log_2 p \times T_{msg}(s) \tag{6}$$

where, $T_{msg}(s)$ is the transmission time for a message of s bytes. The message contains just a few bytes of data representing the center of mass position and total mass of the domain and therefore, s is generally a small number.

3.2.2 Communication

The formation of the *le-tree* accounts for most of the communication that takes place in each iteration (see Figure 3). Corresponding to each of the $\log_2 p$ spatial bisectors, each processor identifies its own data needed by processors on the other side of the bisector and exchanges the data with its "partner" on the other side of the bisector. Each processor, therefore, sends and receives $\log_2 p$ messages.

The first exchange involves at most $(\bar{n} \times N_{byte})$ bytes where, $\bar{n}$ is the average number of particles per processor and N_{byte} is the number of bytes required to represent a particle. In subsequent exchanges, processors send their own data and the data acquired from other processors during previous exchanges. Therefore, successive messages at most double in length. The kth exchange involves at most $(2^k \times \bar{n})$ particles and therefore, the messages are $(2^k \times \bar{n} \times N_{bytes})$ atmost bytes long.

Further, It is reasonable to assume that the time to search the tree for identifying exportable data is proportional to the number of particles searched and merging the received particles into the tree takes time proportional to the number of particles received. We use

$t_{search-p}$ and $t_{merge-p}$ to denote the search time per particle and merge time per particle, respectively.

$$\begin{aligned} T_{le-tree} &= \sum_{k=0}^{\log_2 p-1}[\text{Search Time} + \text{Message Time} + \text{Merge Time}] \\ &= \sum_{k=0}^{\log_2 p-1}[(t_{search-p} + t_{merge-p}) \times 2^k \times \bar{n} + T_{msg}(2^k \times \bar{n} \times N_{bytes})] \end{aligned} \tag{7}$$

3.2.3 Computation

The total computation in the algorithm takes place in three steps - force evaluation (T_{force}), update (T_{update}) and local tree management (T_{local}). The imbalance in workload among processors is due to the dynamics of the simulation problem. As the simulation proceeds, particles move across processor domains causing an imbalance in the number of particles per processor (given fixed processor domains) and the computational load at each processor. However, since the force computation dominates each iteration, it is reasonable to assume that the imbalance in the other computation terms (T_{update} and T_{local}) is negligible. The imbalance in force computation is reflected in the imbalance parameter β. Thus, the average computation time per iteration can be written as:

$$T_{comp} = \beta T_{force} + T_{update} + T_{local} \tag{8}$$

The force evaluation phase is the most time consuming phase of each iteration. For each particle, the tree is traversed from the root and the force on the particle due to other particles or groups of particles are computed. Due to the hierarchical nature of the algorithm, each particle has, on the average, $O(\log N)$ force evaluations associated with it. Let the average number of evaluations per particle be $K_{eval} \log N$, where K_{eval} is a constant that depends on the particle distribution and on the user defined accuracy/tolerance parameter, θ (see figure 1). The time for the force evaluation phase can be expressed as:

$$T_{force} = \bar{n} \times K_{eval} \log_2 N \times t_{eval} \tag{9}$$

where, t_{eval} is the time for each force evaluation[3].

The update phase involves scanning through the list of local particles and updating their position and velocities based on the resultant force on each particle. The time for the update operation is proportional to the number of particles present in the processor.

$$T_{update} = \bar{n} \times t_{update-p} \tag{10}$$

[3]This operation involves evaluating the Newton's force formula. Depending on the accuracy desired, it could be simple first order calculation or may involve a more complex evaluation based on higher order moments. A simple first order calculation is used in our implementation

Table iii: Message System Parameters

Parameter	Definition	Measured Value	
$MAXML$	Maximum fragment length	4096	bytes
t_{start}	Packaging startup time	3.4	ms
$t_{pack-byte}$	Packaging cost per byte	0.9	μs
$t_{latency}$	Latency through network	3.0	ms
t_{rate}	Delay per byte (transmission rate)	1	μs
$t_{unpack-byte}$	Unpackaging cost per byte	0.9	μs

Forming a new tree of local particles and computing their center of mass is essentially a $O(n \log n)$ operation, where n is the number of local particles. The time for this phase, T_{local} can be expressed as:

$$T_{local} = \bar{n} \log_2 \bar{n} \times t_{local-p} \tag{11}$$

4 N-body Simulations on the PVM system

In this section, the performance model presented above is evaluated and compared to measured values. In this experiment, the N-body simulation algorithm was implemented on a network of workstations.

The algorithm was implemented on a system consisting of eight SUN/SPARC workstations connected by a standard ethernet. The PVM programming environment was used to develop our parallel implementation. PVM (Parallel Virtual Machine)[13] is a programming environment, developed at the Oak Ridge National, Oak Ridge, Tennessee, which enables a collection of independent workstations connected by some interconnection network to be viewed as a single concurrent computing resource. PVM has been gaining popularity in research and academic circles mainly due to its flexibility, portability and its ability to use existing computing resources as a distributed system.

The model parameters, t_{eval}, $t_{update-p}$, $t_{local-p}$, t_{search} and t_{merge} are all dependent on the computation capacity of the processors in the system. These constants were measured (see Tables (ii) and (iii). over a large number of trial runs. K_{eval} is a function of simulation parameters such as the tolerance, θ and the initial distribution of particles.

The message transmission time, $T_{msg}(s)$ is the total time to transmit a message of s bytes from one processor to another. This includes the actual transmission delay in the network and the packaging, unpackaging and other processing delays at the two end points. The traffic on the ethernet and the processor loads effect the message transmission times. To minimize the effect of these factors, all our measurements were taken when the system was

lightly loaded. The message system parameters were obtained by sending a large number of messages between two processors in the network and measured the packaging, unpackaging times and message delays for each message. The message transmission time can be expressed as the sum of these three components.

$$T_{msg}(s) = T_{pack}(s) + T_{delay}(s) + T_{unpack}(s) \tag{12}$$

where, $T_{pack}(s)$ is the processing time to package a message of s bytes at the transmitting end, $T_{delay}(s)$ is the time for the message to traverse the network, and $T_{unpack}(s)$ is the processing time to unpackage the message of length s bytes at the receiver end. Based on experiments under the PVM programming environment and the ethernet interconnection of our workstation network, terms in the above equation can be expressed as:

$$T_{pack}(s) = \lceil \frac{s}{MAXML} \rceil \times t_{start} + s \times t_{pack-byte} \tag{13}$$

$$T_{delay}(s) = t_{latency} + s \times t_{rate} \tag{14}$$

$$T_{unpack}(s) = s \times t_{unpack-byte} \tag{15}$$

where, $\lceil . \rceil$ denotes the ceiling function and *MAXML*, the maximum fragment length, is a parameter used by PVM for packaging data for transmission.

The processor sending the message, packages the data in fragments of length *MAXML* and sends it out on the network. The delay through the network involves an initial latency, $t_{latency}$, and a delay, $s \times t_{rate}$, that depends on the transmission (bit) rate of the message system. Unpackaging time is proportional to the length of the message.

Using expressions for message transmission times and the model derived in the previous section, the performance of the N-body simulation algorithm can be predicted and compared with measured results. The speedup obtained on p processors is defined as the ratio of the time per iteration on a uniprocessor over the time per iteration on p processors.

$$\text{Speedup}(p) = \frac{T_{iter}(1)}{T_{iter}(p)}$$

In Figure 6, the speedup versus number of processors is plotted for 128 and 1024 particle systems. These are simulations of a slowly evolving system where the workload remains roughly balanced ($\beta = 1$) throughout the run. The model predicted performance within 12% of measured values with the error being due principally to the β approximation and to the somewhat unequal computational capabilities of the processors.

Figure 7 shows the measured and model speedup for a highly dynamic 1024-particle simulation with a different initial distribution. In this case, the measured workload imbalance

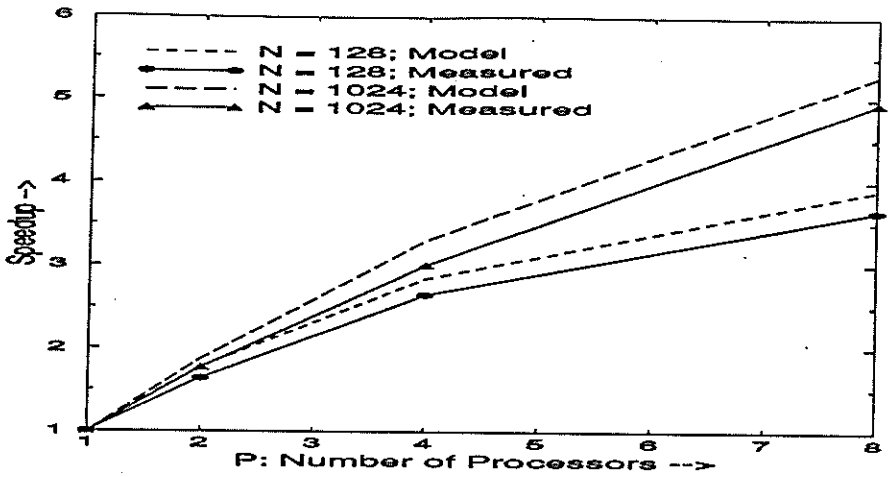

Figure 6: Measured and Model speedups: Slow Dynamic Case
(N: Number of particles simulated. $\beta = 1$)

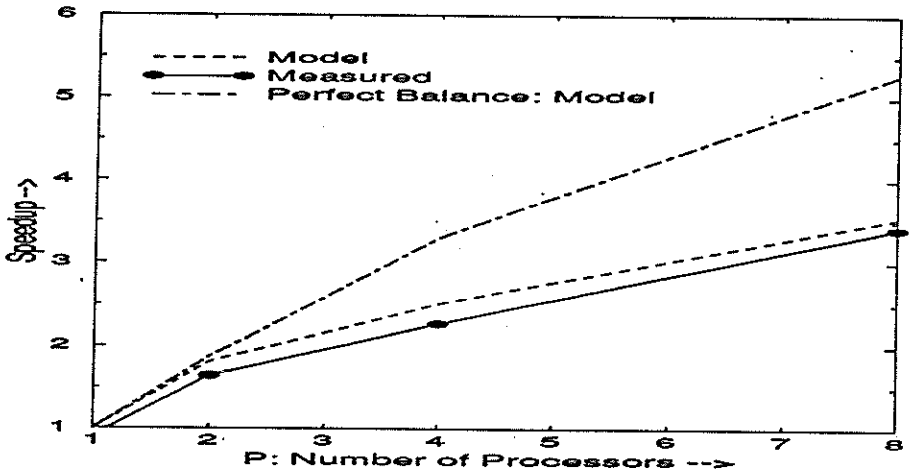

Figure 7: Measured and Model Speedups: Fast Dynamic Case
(N: Number of particles simulated = 1024. $\beta > 1$)

ranges from 1 to 1.2 for two processors, 1 to 1.98 for four processors and 1 to 2.57 for eight processors (Performance model uses average measured β values). Due to the imbalance, the speedups obtained are much smaller than the obtained with the slow dynamic example. Our model predicted performance within 15% of measured values.

Table (iv) lists the error between measured values and those predicted using our performance model. The mean (absolute) error and the maximum error in estimating synchronization, communication and computation times for both examples (Figures 6 and 7) are given. In both cases, the mean error is less than 15% with a lower error for the slow dynamic case, where more accurate approximations of β (and thus for computation times) are used. The errors in synchronization and communication times are mainly due to the effects of network traffic.

Table iv: Errors in Model Predictions

	Slow Dynamic Simulation		Fast Dynamic Simulation	
	Mean Error	Max. Error	Mean Error	Max. Error
Sync. Time	7.3%	10%	8.2 %	12%
Comm. Time	3.2%	3.7%	3.5%	4.1%
Comp. Time	7.05%	13.3%	11.2%	19.6%
Total Time	5.3%	11.6%	7.3%	15.1%

5 Load Balancing

As indicated in the prior section, improper load balancing adversely effects performance. Thus, while the initial load balance may be near perfect, due to the dynamic nature of the N-body simulation, workload distribution among processors may change considerably as the simulation proceeds. Figure 7, for example, indicates that about 40% improvement can be theoretically obtained if ideal load balance can be achieved. This section presents two dynamic load balancing algorithms and examines their performance.

While a load balancing algorithm is expected to reduce workload imbalance, β. there is a cost incurred in executing the load balancing algorithm itself. The load balancing cost per iteration, T_{load}, is defined as the computation and communication time per iteration spent in executing the load balancing algorithm. In general, the load balancing cost has two components: (1) cost for collecting system state information (e.g., load average over all processors) and (2) cost for distributing load based on the system state information.

In our implementation, the load average is computed in the synchronization phase. Since information on processor loads is distributed along with the center of mass information, there is negligible added cost associated with this activity.

The cost for distributing load depends on:

(a) the load balancing algorithm used

(b) how often the algorithm is activated (dependent on the dynamics of the simulation)

(c) the communication and computation parameters of the system.

If the load balancing algorithm results in a new imbalance factor, $\hat{\beta}$, the time per iteration with load balancing can be expressed as a modification of equation (5) as:

$$T_{iter} = T_{sync} + T_{le-tree} + \hat{\beta} \times T_{force} + T_{update} + T_{local} + T_{load} \qquad (16)$$

In this section, two load balancing algorithms tailored to the N-body simulation problem are presented and evaluated. They are referred to as the (1) Domain redistribution and (2) Distributed load balancing algorithms.

5.1 Domain Redistribution (DR)

In the original algorithm presented, the ORB technique described in section 2.2 generates processor domains with equal workload, and those domains remain for the entire simulation. With the domain redistribution(DR) algorithm, the ORB decomposition routine is invoked whenever the load imbalance, β, exceeds a pre-defined threshold parameter, $\beta_{threshold}$. In this way, balance is maintained within certain bounds. The algorithm is simply:

If $(\beta \geq \beta_{threshold})$
    Call ORB algorithm.

The cost associated with this approach can be modeled as follows: Let T_{ORB} be the time taken for ORB decomposition operation. T_{ORB} consists of the following components:

- T_{pool}: Time to "pool" together particles on all processors. The particle information is sent up the processor tree. This involves $\log_2 p$ messages with each successive message doubling in length.
- T_{bisect}: At each level of the tree, bisect the domain into two halves with equal workload. This is analogous to the tree formation operation in step 5 described in section 3.2.
- T_{distr}: At each level, send particle information to processors at the next tree-level. This is similar in complexity to T_{pool}.

Since each node at level i of the ORB-tree has, on the average, $2^i \times \bar{n}$ particles, we have:

$$\begin{aligned} T_{ORB} &= T_{pool} + T_{bisect} + T_{distr} \\ T_{pool} &= \sum_{i=0}^{\log_2 p - 1} T_{msg}(2^i \times \bar{n} \times N_{bytes}) \\ T_{bisect} &= \sum_{i=0}^{\log_2 p - 1} [t_{local-p} \times 2^i \bar{n} \log_2(2^i \bar{n})] \\ T_{distr} &= \sum_{i=0}^{\log_2 p - 1} T_{msg}(2^i \times \bar{n} \times N_{bytes}) \end{aligned} \tag{17}$$

The parameters are defined in Table (ii).

We now proceed to estimate how often the algorithm is activated. To do this, we assume that the imbalance, β, with no load balancing, increases steadily from 1 to a maximum value, β_{max} (Our measurements indicate that this assumption is reasonable) for a total of

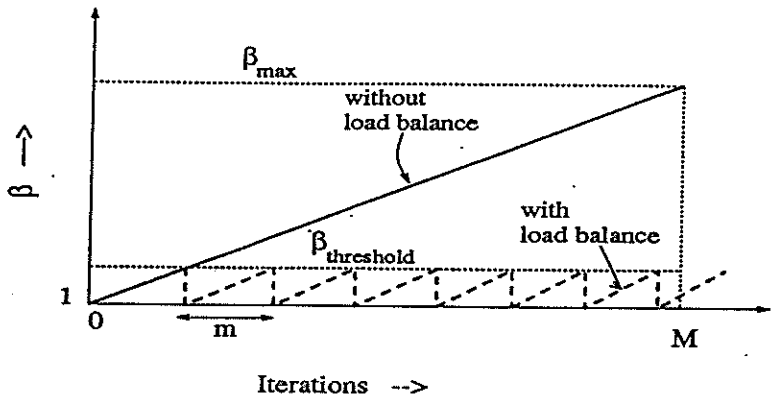

Figure 8: Variation of β

Table v: Performance of Domain Redistribution (DR) Algorithm

p	β	β_{max}	$\hat{\beta}$	$T_{load}(ms)$	Speedup (no balance)	Speedup (with balance)	% gain with load balance
1	1	1	1	0	1	1	0%
2	1.04	1.2	1.03	139.309	1.81	1.79	- 1.1%
4	1.41	1.98	1.03	513.029	2.51	2.87	14%
8	1.8	2.57	1.04	949.068	3.54	3.79	7%

M iterations. The load balancing algorithm is activated whenever the imbalance exceeds $\beta_{threshold}$. Assuming that the linearity of β holds with load balancing, the load balancing algorithm is activated every m iterations, where,

$$m = \frac{(\beta_{threshold} - 1)}{(\beta_{max} - 1)} \times M$$

(see Figure 8). Therefore, the cost per iteration, T_{load}, for the domain redistribution algorithm is given by:

$$T_{load} = \frac{T_{ORB}}{m} \tag{18}$$

Table (v) gives the algorithm performance (in terms of speedup) as predicted by the model for the 1024-particle fast dynamic system (see Figure 7). β and β_{max} are measured values for the no-balance case. The threshold parameter, $\beta_{threshold}$, was taken as equal to 1.05, and the total number of iterations, M, was set equal to 1000. From the data in Table (v), it can be seen that DR algorithm (although achieving good balance) does not yield good overall performance due to its high execution costs. The measured speedups were close to the speedups predicted by the model and are plotted in Figure 10.

At Each Processor:
    If (my_load $\geq$ *HWM*)
        Shrink Domain by $\frac{\text{my\_load - } HWM}{HWM}\%$
        Send all particles outside my domain to neighbors
    If (particles Received)
        Expand domain to accommodate them.

Figure 9: Distributed Load Balancing Algorithm

5.2 Distributed Load Balancing (DLB)

In the DR algorithm above, all the processors are involved in the load balancing operation even though only a few processors may be heavily loaded. An alternate approach is for the heavily loaded processors to execute the load balancing algorithm locally and balance their workloads. In the distributed load balancing (DLB) algorithm, the heavily loaded processors "shrink" their domains by sending particles to their neighbor processors which, in turn, "expand" their domains. The processors reach a new distribution over a number of iterations where the heavily loaded processors have smaller domains and the lightly loaded processors have larger domains and the workload is balanced.

The DLB algorithm is given in Figure 9. *HWM*, the High Water Mark, is the threshold load at which a processor should send out particles and is typically set to 10 to 20% above the system average. The cost associated with this load balancing technique is due to:

- Computation cost: Cost for comparing local load with the *HWM*, shrinking the local domain and identifying particles to be sent. This cost is principally proportional to the number of particles present in the processor domain.
- Communication cost: Cost of sending the particle to appropriate destination processors. Since processors do not wait for messages, the transfer of particles is assumed to be overlapped with other computation and these costs are therefore assumed to be negligible.

The cost per iteration for the distributed load balancing algorithm can be expressed as:

$$T_{load} = t_{lb} \times \bar{n} \tag{19}$$

where, t_{lb} is the time taken to test whether a particle is outside a processors domain and needs to be sent out. For our implementation, $t_{lb} = 0.02$ ms. The load balancing cost is, therefore, very small compared to the total time per iteration.

Table vi: Performance of Distributed Load Balancing (DLB) Algorithm

p	β	β_{max}	$\hat{\beta}$	T_{load} (ms)	Speedup (no balance)	Speedup (with balance)	Gain with load balance
1	1	1	1	0	1	1	0%
2	1.04	1.2	1.03	10.24	1.81	1.82	0.5%
4	1.41	1.98	1.1	5.12	2.51	3.07	23%
8	1.8	2.57	1.32	2.56	3.54	4.41	25%

Since load balancing activity is localized around heavily loaded processors, if all processors in a certain neighborhood are heavily loaded, some amount of "thrashing" may occur. As a result, the DLB algorithm achieves poorer balance than the DR algorithm. However, though the DLB algorithm achieves poorer balance, it results in improved speedups due to its relatively low execution costs. For the 4 and 8 processor cases a significant improvement of about 25% is achieved. Table vi lists the values of β, T_{load} and speedup evaluated using our model for a 1024-particle simulation with HWM= 1.1. (β and β_{max} are measured values for the no-balance case reported in Figure 7. $\hat{\beta}$ is the measured imbalance for the distributed load balance case.). The measured speedups for the distributed load balancing algorithm are similar to the model predictions and are plotted in Figure 10.

In Figure 10, the performance of the load balancing schemes are compared against the no-balance and the perfect balance cases. Speedup for the perfect balance case is evaluated using the performance model (equation 16) with the imbalance, set to 1 and load balancing cost, T_{load} set to zero. The error in model predictions are fairly small (less that 15%) and are mainly due to the approximation of $\hat{\beta}$ in the DR algorithm and the approximation of T_{load} for the DLB algorithm.

6 Summary, Conclusions and Future Work

In this paper, we have presented a parallel N-body simulation algorithm, based on the Barnes-Hut algorithm, to study the evolution of a system of particles under gravitational forces. The algorithm can be generalized to a wide variety of applications. N-body techniques are good candidates for use as parallel benchmarks and are prototypical of a wide class of algorithms referred to as synchronous iterative algorithms. We have implemented the algorithm on a network of workstations connected over a standard ethernet and intend to use the algorithm as part of a benchmark suite oriented towards evaluation of distributed memory parallel processors.

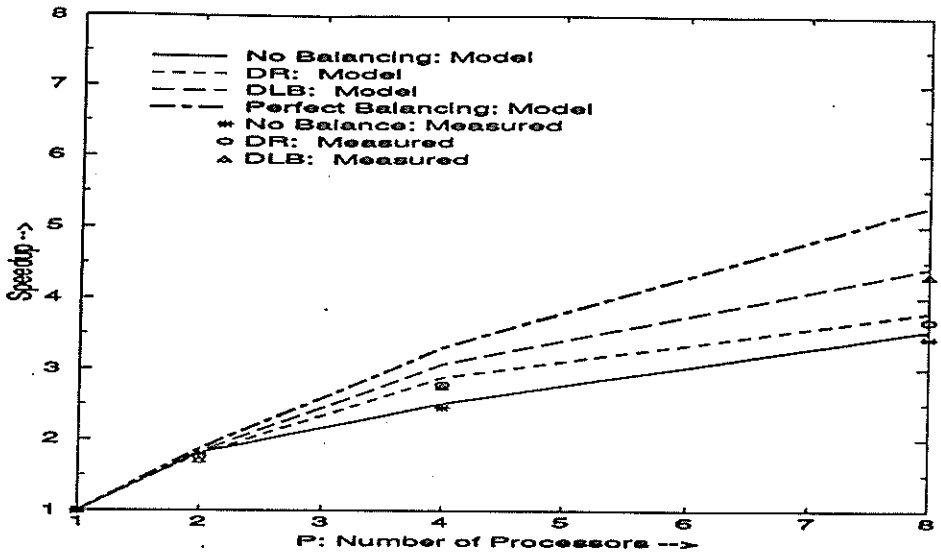

Figure 10: Comparison of Load Balancing Schemes

Performance models can be used to estimate the performance of the algorithm on a given system, identify performance bottlenecks, and to evaluate possible enhancements/modifications to the system and to the algorithm. We have outlined a framework for developing simple performance models for a class of synchronous iterative algorithms. A mean-value performance model is developed for the N-body simulation algorithm and the model is used to estimate the performance of the algorithm on a network of workstations. The performance predicted by our model compared very well (within a 15% error) with the measured performance.

Based on our model, several performance bottlenecks have been identified and suggestions made that may improve overall performance . One improvement which increases processor utilization results from maintaining a balanced workload over all processors. This reduces the total execution time and, therefore, improves the speedup obtained. In this paper, two load balancing algorithms are presented and models developed to estimate their performance. These algorithms were also implemented on our workstation network. The distributed load balancing technique yielded up to 25% improvement in speedup. The measured performance and model prediction matched within a 15% error. Thus the model is a reasonable predictor of performance and use of the DLB algorithm can yield substantial performance improvements.

We are currently looking into techniques that can mask communication latencies in distributed memory systems for a class of iterative algorithms. Other enhancements we are investigating relate to the use of high speed ATM based communication networks and associated protocols to reduce the communication time per iteration and hardware based synchronization networks and related mechanisms to reduce the synchronization time. Performance

improvements due to these enhancements can be evaluated using the model presented in this paper and thus provides us with a framework to quantify the performance implications of these design decisions.

References

[1] Andrew W. Appel. An Efficient Program for Many-body Simulation. *SIAM Journal of Scientific and Statistical Computing*, 6, January 1985.

[2] Joshua E. Barnes and Piet Hut. A Hierarchical O(N log N) Force Calculation Algorithm. *Nature*, 324(4), Dec. 1986.

[3] Roger Chamberlain and Mark A. Franklin. Hierarchical Discrete-Event Simulation on Hypercube Architectures. *IEEE MICRO*, August 1990.

[4] Edward A. Desloge. *Classical Mechanics.* John Wiley, New york, 1982.

[5] M. Dubios and F.A. Briggs. Performance of synchronized iterative processes in multiprocessor systems. *IEEE Trans. Software Eng.*, (4), July 1982.

[6] G.C. Fox et al. *Solving Problems on Concurrent Processors.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[7] P. Hanrahan et al. A Rapid Hierarchical Radiosity Problem. *Proceedings of SIGGRAPH*, 1991.

[8] Leslie Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems.* ACM Press, 1987.

[9] K. Hwang and F.A.Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, NY, 1986.

[10] Jaswinder Pal Singh, John L. Hennessey and Anoop Gupta. Implications of Hierarchical N-body Methods for Multiprocessor Architecture. Technical Report CSL-TR-92-506, Computer Systems Laboratory, Stanford University, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, 1992.

[11] John Salmon and Michael S. Warren. Astrophysical N-body Simulations on the Delta. Technical report, California Institute of Technology, April 1992.

[12] Harold S. Stone. *High-Performance Computer Architecture.* Addison-wesley Pub. Co., Reading, Mass., 1990.

[13] V.S.Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2, Dec. 1990.