A Load Balancing Package on Distributed Memory Systems and its Application to Particle–Particle Particle–Mesh (P3M) Methods *

X. Yuan[1], C. Salisbury[1], D. Balsara[2] and R. Melhem[1]

[1] Department of Computer Science,
The Univ. of Pittsburgh
and
[2] The Nat. Center for Supercomputing Applications,
The Univ. of Illinois at Urbana-Champain.

Abstract

We present a tool, *Bisect*, for balanced decomposition of spatial domains. In addition to applying a nested bisection algorithm to determine the boundaries of each subdomain, Bisect replicates a user specified zone along the boundaries of the subdomain in order to minimize future interactions between subdomains. Results of running the tool on the Cray T3D system using both shared memory operations and MPI communications are reported and discussed. In addition, Bisect is used in a parallel implementation of a particle–particle/particle–mesh (P3M) simulation program on the Cray T3D system. The performance of the P3M program with different load–balancing criteria is evaluated and compared. The results show that the use of the Bisect package balances the load efficiently and minimizes communication on the T3D massively parallel system.

1 Introduction

A wide range of scientific applications suffer from load imbalances and excessive interprocessor communications when they are ported for execution on massively parallel processing systems. In this paper, we will introduce a dynamic load balancing package, Bisect, which we developed on the Cray T3D system and which has been ported to the IBM SP-2 and to workstation clusters using MPI primitives. The package uses the *nested bisection* method to achieve dynamic load balancing for applications in which computations are non-uniformly distributed over some physical domain [6]. Specifically, given N objects, each associated with a position in a physical domain, Bisect decomposes the domain into disjoint subdomains. Each subdomain, along with the objects it contains, is mapped to a different processor. The goal is to choose a decomposition that will balance the computation among the processors in the system. The decomposition can be done dynamically to rebalance the workload after objects move between subdomains.

*This work is in part supported by an NSF Grand Challenge grant ASC-9318185

In addition to the load balancing feature, the Bisect package also adopts a technique called *domain extension* to optimize the communication in applications which exhibit local spatial dependencies. In such applications, the computation associated with an object not only depends on the attributes of that object, but also on the attributes of the objects that are spatially close to it. Bisect has the capability of extending the subdomain assigned to each processor to contain those objects which are not in that subdomain, but affect the computation associated with it. These objects are called *ghost objects*.

The *nested bisection* technique and the *domain extension* technique are combined seamlessly in the Bisect package to achieve good performance. This package can be used for a large class of scientific applications which exhibit *spatially dependent fine-grained parallelism*. Examples include Molecular Dynamics and N-body simulations [7, 8], Finite Elements and Finite Difference methods on irregular grids, Algebraic Multigrid computations, and particle-particle/particle-mesh (P^3M) methods [1, 4, 5, 7].

To demonstrate the load–balancing capability of Bisect, we apply it in a parallel implementation of the P3M method. The P3M method is an efficient N-body simulation method which splits the interparticle forces into a rapidly–varying short range part and a slowly–varying long range part. The particle–particle method, PP, is used to find the short range contribution to the force on each particle, and the particle–mesh method, PM, is used to find the long range contribution to the force on each particle. In this paper, we will discuss how Bisect reorganizes the particles to facilitate the efficient computation of both the PP and the PM forces.

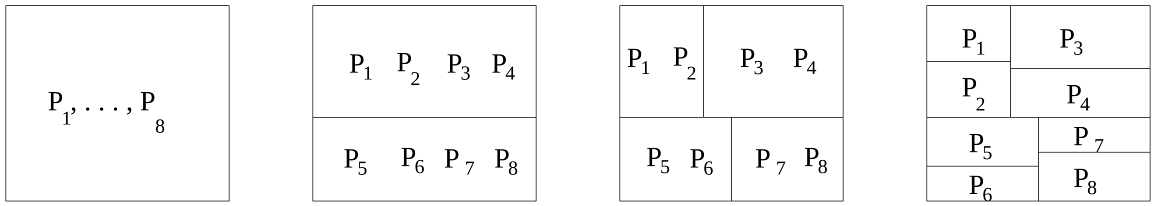
The rest of the paper is organized as follows. Section 2 presents the fundamental concepts of nested bisection and domain extension. Section 3 introduces the Bisect package. Section 4 reports the performance evaluation for Bisect. Section 5 discusses the methods to incorporate Bisect in a P3M program to efficiently reorganize the particles in the system, and presents the performance of a parallel P3M implementation. Section 6 concludes the paper.

2 Parallel Nested Bisection with Domain Extension

Consider N objects, each with a number of data attributes, distributed over a three-dimensional, rectangular, physical domain. One of the attributes may be a weight set to reflect the amount of computation needed to process that object. The Bisect package runs on a distributed memory system of p processors, where p is power of 2. It assumes that each processor initially stores $n = \frac{N}{p}$ objects distributed over the entire physical domain. When invoked, Bisect uses a nested bisection algorithm [2] to redistribute the objects among the processors such that:

- each processor is assigned objects that fall in a rectangular 3-dimensional subdomain, and
- the sum of the weights of the objects assigned to each processor is almost the same for all processors.

In the first iteration of the bisection algorithm, the domain is decomposed into two subdomains so that the difference between the sums of the weights of the subdomains is as



(a) initial distribution (b) after iteration 1 (c) after iteration 2 (d) after iteration 3

Figure 1: Domain decomposition among 8 processors

small as possible. Then, the same process is applied to the two subdomains in parallel, and the process is repeated, recursively, for $\log p$ iterations. In other words, during iteration i, $1 \leq i \leq \log p$, the p processors are grouped into 2^{i-1} groups of $\frac{p}{2^{i-1}}$ processors each. At the beginning of the iteration, the problem domain is already partitioned into 2^{i-1} subdomains and the objects in each subdomain are stored in a single group of processors. At the end of the iteration, each processor group is divided into two groups, and the corresponding subdomain is divided into two subdomains, with the objects in one subdomain residing in one half the processors and the objects in the other subdomain residing in the other half of processors. Figure 1 shows an example of the correspondence between processors and subdomains when a 2-dimensional domain is divided among eight processors.

In order to be more specific, consider objects belonging to a subdomain, S that are distributed among the processors of a group of processors, G. The following three steps should be accomplished during iteration i:

1. Determine how to divide S into two subdomains, S_1 and S_2, and G into two groups G_1 and G_2.

2. Partition the objects in S into two subsets, one belonging to S_1 and one belonging to S_2,

3. Move the objects in S_1 to the processors in G_1 and the objects in S_2 to the processors in G_2.

Lemma 1: If S_p is the execution time for a p processor nested bisection algorithm, where p is a power of two, then the speedup of the algorithm is bounded by $\frac{S_2}{S_p} = O(\frac{p}{\log p})$.

Proof: let N be the total number of objects in the system. During each iteration of the nested bisection algorithm, ignoring the work needed to determine the subdivision of domains, each object has to be visited at least once to determine the subdomain to which it belongs. Hence, the total amount of work in each iteration of the nested bisection algorithm is $O(N)$. For p processors to execute the Bisect algorithm, $log(p)$ iterations are needed. The total amount of work for p processors running the Bisect algorithm is thus $O(N \times log(p))$. Therefore, assuming perfect load balancing, $S_p = \frac{O(N \times log(p))}{p}$ and $S_2 = \frac{O(N \times log(2))}{2}$. Thus, $\frac{S_2}{S_p} = O(\frac{p}{\log p})$. □

To reduce interprocessor communications in application programs, a *domain extension* technique is used in Bisect. In domain extension, each processor gets not only the "real objects" which are in the domain assigned to it, but also the "ghost objects" which are not in that domain but contribute to the computations performed on the "real objects". In other words, each processor gets objects in its own domain and copies of the objects that are surrounding the domain, within a user specified distance. This way, all the information needed to complete the computation in a processor is stored locally. Once the objects are redistributed by Bisect, no further communication is needed during the computation.

By incorporating the domain extension into the nested bisection, the package is able to achieve both load balancing and optimization of the communication in the application. Our experimental results on both randomly generated objects and heavily clustered objects show that Bisect is a very efficient package both in terms of execution time and the effect on load balancing.

3 The Bisect Package

Although the main objectives of Bisect are to redistribute the objects so that the computational load is balanced in each processor and to obtain the extended domain, Bisect provides many features that give the user maximum flexibility. Specifically, in addition to specifying the domain and the problem size in terms of the number of objects and the number of attributes for each object, the user may specify the following:

- A bin size which affects the quality of the load balancing algorithm, as described later,
- A choice among the following load balancing criteria: 1) balance the domain size; the resulting subdomains will have the same size. 2) balance the number of objects; the number of objects in the resulting subdomains will be balanced. 3) balance the weights; the sum of the weights in the resulting subdomains will be balanced.
- The amount by which each subdomain is to be extended.
- A choice of periodic and non-periodic boundary conditions for extending the exterior boundaries of the domain.
- The sequence of dimensions that are to be bisected, or a default automatic bisection. The automatic bisection results in the domain being bisected in each dimension in a round-robin manner. If the user has some specific knowledge of object distribution, (s)he can manage the bisected dimensions to achieve the best load balancing results.
- A dynamic balancing mode which can be used after Bisect is called at least once. In this mode, the boundaries of the subdomains are not recomputed. The particles and ghosts are redistributed using the boundaries computed in the previous call to Bisect.

Assume that, before the i^{th} iteration of Bisect, $1 \le i \le \log p$, the objects in a given subdomain are distributed among processors $P_1, \ldots, P_k$, where $k = \frac{p}{2^{i-1}}$. The process to carry out a bisection of the subdomain along a specific dimension with domain extension is divided into three parts:

1) Determine the bisection line: A parallel sorting algorithm is used for this task. While an exact sorting along the dimension to be bisected can find the optimum bisection line, it introduces a high overhead of $O(N \log N)$ for sorting and counting. For efficiency considerations, we use a parallel bin sorting algorithm where the dimension to be bisected is divided into a certain number of bins (slices). Each processor sums the weights of its local objects in each bin, and a global sum algorithm is used to obtain the total weight of the objects in each bin. This information is then used by each processor to determine the bisection line along a bin boundary. Note that the time complexity of the bin sorting algorithm is $O(N + \log p)$, which greatly reduces the sorting overhead. While bin sorting restricts the cut line to be along bin boundaries and therefore may introduce a load imbalance, our performance study shows that increasing the number of bins in the system to improve load balancing does not significantly degrade the runtime performance of Bisect. Acceptable performance in terms of running time and load balancing can be achieved by using reasonably large number of bins.

In some scientific applications, the distribution of objects is extremely non-uniform. For these applications, the load imbalance incurred by the bin sort may cause some processors to get a subdomain of size zero. This may cause runtime errors in the original application, since application programs generally assume non-null physical domains. Precaution is taken to prevent this situation; because the number of bisections along each dimension is known, Bisect can guarantee that each processor will get a subdomain which contains at least one bin in each dimension.

2) Partition the objects: Once the bisection line is decided, each processor determines how to distribute the objects it has such that the objects belonging to one of the two subdomains are in processors $P_1, \ldots, P_{\frac{k}{2}}$ and the objects belonging to the other subdomain are in processors $P_{\frac{k}{2}+1}, \ldots, P_k$. Specifically, a processor P_j examines each of its "real objects", and determines one of the following:

1. to keep the object as a real object,
2. to send the object to processor $\bar{P}_j$ as a real object, where $\bar{P}_j = P_{j-\frac{k}{2}}$ or $P_{j+\frac{k}{2}}$ depending on whether $j \leq k/2$ or $j > k/2$, respectively.
3. to keep the object as a real object and send it to $\bar{P}_j$ as a ghost object, or
4. to keep the object as a ghost object and send it to $\bar{P}_j$ as a real object.

In other words, when a cut is placed along a dimension, an internal boundary condition is created. All the objects residing within the user-specified distance of the boundary are duplicated as real objects in one processor and as ghost objects in another processor. Similarly, P_j examines each of its "ghost objects" and determines whether it should keep the object, send it to $\bar{P}_j$, or both. This way all the objects in the extended domain will be moved to the proper processors.

Besides the internal boundary conditions, some applications require periodic or wrap-around boundary condition for the entire problem domain. The objects in one side of the initial domain will affect the computation of the objects in the opposite side of the domain. As a result, the objects in the boundary bins of the initial problem domain must be duplicated.

3) Move the Objects: After the second phase, each processor, P_j, packs all the objects that it should send to $\bar{P}_j$ in a message and sends that message.

In the following two Lemmas, we establish some bounds on the memory requirements in each of the p processors running Bisect, assuming that each processor initially has $n = \frac{N}{p}$ objects and that all objects have equal weights.

Lemma 2: Without domain extension, the maximum memory requirement for the nested bisection algorithm in each processor is $O(n\sqrt{p})$.

Proof: Let m be the maximum number of objects in a processor before any iteration. After the iteration, the maximum number of objects in that processor is at most $2m$. Since each processor initially has n objects, after i iterations, the number of objects in a processor is at most $2^i n$. On the other hand, after i iterations, each group of $\frac{p}{2^i}$ processors has at most $\frac{N}{2^i}$ objects. Hence, a processor has at most $\max\{\frac{pn}{2^i}, 2^i n\}$ objects after the ith iteration. The maximum with respect to i is obtained when $2^i n = \frac{pn}{2^i}$, which is when $i = \frac{log(p)}{2}$. Hence the maximum number of objects in each processor during the execution of Bisect is less than or equal to $2^{\frac{log(p)}{2}} n = n\sqrt{p}$. $\Box$

From Lemmas 2, we can see that in the worst case, the memory requirement for Bisect may be quite large. Object rebalancing within each group before each bisection step will reduce the maximum memory requirements in each processor from the $O(n\sqrt{p})$ specified in Lemma 2 to $O(2n)$. Such a rebalancing redistributes the $\frac{N}{2^i}$ objects in a processor group among the $\frac{p}{2^i}$ processors in that group. A simple dimension exchange algorithm [3] is adequate for that purpose.

The need for redistributing the objects among the processors in a processor group before each domain bisection is even more important if domain extension is used, since the maximum memory requirements in such cases can only be bound by $O(N)$ as shown in the following lemma.

Lemma 3: With domain extension, the maximum memory requirement for the nested bisection algorithm in each processor is $O(N)$.

Proof: For each processor, an object can either be a real object or a ghost object or an irrelevant object(outside the domain). Since an object can not be both real object and ghost object, each processor can have at most the total number of objects in the system, which is equal to $O(np)$. $\Box$

4 Experimental Results

In this section, we present experimental results for the performance Bisect using objects distributed both uniformly and non-uniformly on three dimensional domains. We show the effect of the number of bins and the number of processors on Bisect's runtime and on the effectiveness of Bisect to achieve load balancing. This effectiveness is measured by the load imbalance ratio, which is defined as the largest percentage difference between any one processor's workload and the average workload. The experiments presented in this section assume that all objects have equal weight. Objects with different weights will be dealt with in the next section.

Table 1 shows the performance of Bisect on uniformly-distributed objects with 1000 bins along each dimension and each subdomain extended by one bin. The program was run on a 32 PE system. The results show that, in all cases, the load imbalance ratio is less than one percent. The algorithm is stable for uniformly distributed objects in the sense that the imbalance ratio is not significantly affected by the number of particles in the system. It also shows that Bisect runs very fast. With 32 Cray T3D processors, it only takes a fraction of a second to decompose a domain with 1 million objects.

Table 1: Performance of Bisect on uniformly distributed objects using 1000 bins

Total no. of objects	no. of objects per PE excluding ghosts				no. of objects per PE including ghosts				Time (sec)
	min	max	ave	imb. ratio	min	max	ave	imb ratio	
131072	4065	4126	4096	0.7%	4145	4213	4176	0.9%	0.10
262144	8135	8246	8192	0.7%	8312	8420	8356	0.8%	0.19
524288	16238	16514	16384	0.8%	16575	16855	16712	0.8%	0.38
1048576	32558	32982	32768	0.7%	33206	33674	33427	0.7%	0.74

Table 2: Performance for different numbers of bins on 524288 uniformly distributed objects

no. of bins	no. of objects per PE excluding ghosts				no. of objects per PE including ghosts				Time (sec)
	min	max	ave	imb. ratio	min	max	ave	imb ratio	
500	16273	16514	16384	0.8%	16925	17164	17044	0.7%	0.39
1000	16306	16462	16384	0.5%	16642	16772	16713	0.4%	0.39
5000	16371	16403	16384	0.1%	16434	16472	16452	0.1%	0.43
10000	16378	16392	16384	0.0%	16404	16435	16420	0.1%	0.47

Table 3: Performance for different numbers of bins on 512000 non-uniformly distributed objects

no. of bins	no. of objects per PE excluding ghosts				no. of objects per PE including ghosts				Time (sec)
	min	max	ave	imb. ratio	min	max	ave	imb ratio	
500	11263	21926	16000	37.0%	17483	53098	33352	59.2%	0.58
1000	12904	18213	16000	13.8%	17084	32639	23913	36.5%	0.49
5000	15585	16425	16000	2.7%	16177	18678	17442	7.1%	0.45
10000	15686	16196	16000	1.2%	16147	17353	16717	3.8%	0.51

Table 2 and 3 show the effect of the number of bins on the performance of Bisect for uniformly and non-uniformly distributed objects, respectively. Here we can see the general pattern that a larger number of bins leads to better load balancing. For uniformly distributed objects, 500 bins in each dimension achieves good load balancing. However, the number of bins in each dimension has a dramatic impact on the load balancing when the objects are distributed nonuniformly. With 500 bins in each dimension, load balancing after Bisect is poor. The balance is much better when 5000 or more bins are used.

It is interesting to observe that the number of bins has a minor effect on the runtime of Bisect. For the uniform case, increasing the number of bins from 500 to 10000 only slows down Bisect by 10%. This is because the number of bins affects only the message length in the global sum process, and this accounts for a fraction of the total Bisect time. For the non-uniform case, increasing the number of bins can reduce Bisect's run time. Specifically, more bins lead to a more uniform distribution of objects during Bisect's execution, which balances the workload of Bisect better. Therefore, using large numbers of bins to achieve better load balancing is always recommended.

Table 4 shows the performance of Bisect on different numbers of processors assuming 512K objects, 1000 bins along each dimension and a 1 bin domain extension. The second and third columns show that the imbalance ratio increases with the increase in the number of PEs. The fifth column shows the speedup over two-processor execution time. The sixth column lists the theoretical upper bound to the speedup of the algorithm which results from Lemma 1. The last column shows the percentage of actual speedup relative to the upper bound. The results show that, up to 128 processors, Bisect achieves 87.4 percent of the achievable speedup.

Table 4: Performance for different number of processors on 524288 uniformly distributed objects

no. of PEs	imb. ratio without ghosts	imb. ratio with ghosts	time (sec)	actual speedup	speedup upper bound($\frac{p}{2\times lg(p)}$)	speedup percentage
2	0.0%	0.0%	1.158	1	1	100.0%
4	0.1%	0.1%	1.163	0.996	1	99.6%
8	0.2%	0.2%	0.880	1.316	1.333	98.7%
16	0.6%	0.6%	0.593	1.953	2	97.6%
32	0.8%	0.8%	0.377	3.072	3.2	96.0%
64	1.0%	1.2%	0.234	4.949	5.333	92.8%
128	1.5%	1.6%	0.145	7.986	9.142	87.4%

We have implemented two versions of Bisect on the Cray T3D. One uses the T3D shared memory operations, and the other uses the standard MPI interface. The version that uses MPI is more portable to other platforms and has also been ported to the IBM SP2 system. All the results presented so far are for the version that uses shared memory operations. In Table 5, we compare the performance of the two versions of Bisect in an experiment that uses 512K objects, 1000 bins along each dimension and a 1 bin domain extension. As expected, using shared memory operations achieves slightly better performance and better scalability, since the MPI routines incur more overhead for communication than shared memory primitives.

Bisect is ideally suited for applications where objects move rapidly through space. In this situation, objects may change subdomains between iterations of the program. Communication is required to move the objects to the processes that hold the subdomain corresponding

Table 5: Share memory operations versus MPI routines

no. of PEs	Share memory operations		MPI routines	
	time(sec)	speedup	time(sec)	speedup
8	0.880	-	0.927	-
16	0.593	1.48	0.630	1.47
32	0.377	1.57	0.421	1.50
64	0.234	1.61	0.264	1.59
128	0.145	1.61	0.173	1.53

to the new location. This object movement can also cause the workload to become imbalanced. The use of Bisect prior to each iteration of the program can handle both tasks.

On the other hand, in some applications objects move slowly, or not at all. These applications may use Bisect to perform the initial object distribution and workload balancing, or to move traveling objects to the processes that contain their new subdomain. The new static nature of object distribution may not require repeated recalculation of the bisection lines. These applications can make use of a Bisect option to reuse the bisection lines determined in the previous call to Bisect.

In order to understand the performance improvement available, we ran experiments in which Bisect is called in four different scenarios. In the first scenario, all three steps of Bisect are needed to move each object from its initial, randomly assigned processor, to the processor that handles the subdomain where the object resides. In the second scenario, the same initial, random distribution of objects to processors is used, but the domains developed during the first call to Bisect is reused. The difference between the first and second scenarios is the time it takes to execute step one. The third scenario leaves the objects in the proper processors, but requests Bisect to redistribute the workload. Thus step one is executed, but only the ghost objects need to be moved during step 3. The difference between this and the first scenario gives some idea of the communication time required to move objects to the appropriate processors. Finally, the fourth call leaves the objects on the correct processors and reuses the existing subdomain boundaries. This scenario represents the computation time for $\log p$ iterations of examining all the particles in step 2, and the communication time to move the ghost particles in step 3. This represents the time that it takes Bisect to execute when the distribution of the objects changes very little or not at all. The difference between this and the third call represents the time to execute step one of Bisect. The difference between this and the second call to Bisect gives some idea of the communication time required to move objects to the appropriate processors.

The results of testing the four scenarios on a problem with 512K uniformly distributed particles are given in Tables 6 and 7 for the shared memory and the MPI implementations, respectively. The data for each number of processors is very consistent in showing the time saved when step one or step three is eliminated. The motivation for eliminating step one comes from a desire to improve performance when the objects do not move very far or very rapidly. When this occurs, there will be very little communication during step three. Bisect performance without bisection line reuse is represented by the third scenario. The improvement due to the reuse of bisection lines is represented by the fourth scenario.

Table 6: Time for Bisect using shared memory operations

Shared Memory Communications								
no. of PEs	1,000 bins				10,000 bins			
	scenario:				scenario:			
	1	2	3	4	1	2	3	4
4	1.45	1.23	1.28	1.08				
8	0.908	0.744	0.781	0.625	0.973	0.739	0.846	0.619
16	0.614	0.503	0.525	0.419	0.678	0.482	0.589	0.413
32	0.389	0.315	0.334	0.263				

Table 7: Time for Bisect using MPI.

Message Passing Interface Communications								
	1,000 bins				10,000 bins			
no.	scenario:				scenario:			
of PEs	1	2	3	4	1	2	3	4
4	1.59	1.37	1.29	1.08				
8	0.994	0.827	0.789	0.630	1.07	0.802	0.859	0.624
16	0.665	0.549	0.535	0.424	0.745	0.545	0.609	0.419
32	0.425	0.346	0.344	0.269				

With 1000 bins, the improvement can be up to 20% for either communication method. Note that increasing the number of bins slightly increases the time to perform step one. This is expected because the step one message size increases with more bins. Thus the improvement from eliminating step one increases as the number of bins increases. Note also that in scenario 4, there is very little difference between MPI communication and shared memory communication. This is because there is very little communication between processors at all, and any differences in run time will not be large.

It is surprising that the time for scenario 3 is roughly the same for both communication techniques. This means that step one takes about the same time in both techniques. We can see that it is step three that is affected by the choice of communication method. Step three performs better with the shared memory operations. One possible explanation for the similar performance of phase one in both the shared memory and the MPI implementations is the need for global synchronization between messages in phase one. The synchronization delays may be greater for the shared memory operations than for the MPI operations, thus masking any advantage to the shared memory communication.

5 Application to a Particle-Particle/Particle-Mesh Simulation

Bisect was used in a Particle-Particle/Particle-Mesh (P^3M) program that was written to study galaxy formation. The fundamental computational problem in particle based study of galaxy formation is to evaluate the forces at each timestep and advance the system in time. There is a close similarity between this problem and problems in plasma physics. The program is based on the algorithm given in Hockney and Eastwood [7] though other algorithms like the one in Balsara and Brandt [1] would also benefit from the strategies developed here. Thus the utility of Bisect extends to almost all forms of plasma physics simulations.

The P^3M algorithm evaluates the gravitational force on a system of self-gravitating particles. It does this by splitting the force into two parts, a long range force and a short range force. The long range force evaluation is an accurate representation of the force contribution to a given particle from all particles that are farther from it than a certain distance, known as the cut off distance. Because a mesh was used, this step of evaluating the long range forces is also known as a particle mesh (PM) step. Because interpolations are involved in the PM step, the force exerted on a particle from other particles that are within the cut off distance is not represented. Thus an extra step has to be put in to supply the remaining force contribution. This is the short range force which is computed between pairs of particles within the cut off distance. It is known as the particle-particle (PP) step.

5.1 Particle–Mesh method

The particle–mesh method computes long range forces in the system by solving the field equations on a mesh. This method has the advantage of low time complexity — pair forces on N particles are computed in $O(N)$ operations rather than $O(N^2)$. The aim of the PM method is to compute the force by solving a linear field equation relating the force to the density of mass. This method works for any problem where the force field is a sum over particles or, equivalently, a linear convolution of the density. The convolution can be performed rapidly in the Fourier domain using the Fast Fourier Transform (FFT) algorithm.

The force calculation in the PM method may be divided into three phases :

1. **Density assignment:** compute the density on the mesh points by interpolating from particle positions.

2. **Force field transformation:** compute the potential and force at the mesh points from the density using Fourier transform techniques.

3. **Force extrapolation:** extrapolate the force back to the particles from the mesh points.

Two fundamental data structures are involved in the PM method. The first is a list of particles. Each particle is characterized by its position and other attributes such as, velocity, density and force. This list is stored as d one dimensional arrays, where d is the total number of attributes. The second data structure is a mesh, called *the PM mesh*. The mesh has the dimensionality of the simulation space, which is equal to 3 in our case. Forces and densities at the mesh points are stored in 3-dimensional arrays. In a parallel implementation on a p-processor distributed memory system, both the PM arrays (which we will call the PM mesh) and the particle list will be distributed among the p processors in the system. In general, however, there is no correlation between the processor storing a particle's position and the processor storing the mesh cell which contains that particle.

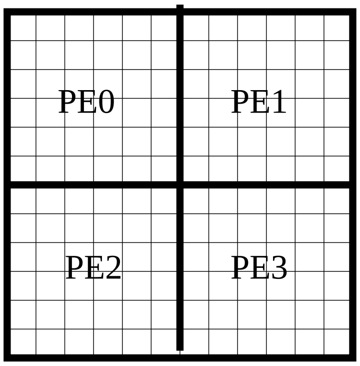


Figure 2: Distribution of the PM mesh on 4 processors

This means that the particle data may need to be moved in the density assignment and the force interpolation phases.

In our implementation, the PM mesh is uniformly distributed over the processors using standard CRAFT block distribution primitives (see for example Figure 2). Depending on the distribution of the particles, it may be the case that some mesh cells have many more particles than do others. This could present a load balancing problem. Bisect can be used for the PM method to achieve load balancing by partitioning the domain based on the number of particles in each domain rather than on the number of mesh points in each domain. Furthermore, it also optimizes the inter–processor communication when the domain extension technique is used, as described next.

Density Assignment

This phase calculates the densities at the grid points of the PM mesh by interpolating particle masses. The second–order Triangular Shaped Cloud (TSC) scheme [7] is used for the interpolation. In this scheme, a particle's mass is spread over a cube in three dimensions with 27 grid points centered on the *nearest grid point* (NGP). The weight given to each of these points is the product of three weights, one for each dimension. For a particle whose first coordinate is x (in units of the PM mesh spacing), the corresponding NGP index is $I = \lfloor x + \frac{1}{2} \rfloor$. The weights assigned to I and $I \pm 1$ are

$$W_I(x) = \frac{3}{4} - (x - I)^2, \quad W_{I\pm1}(x) = \frac{1}{2}(x - I \pm \frac{1}{2})^2$$

The simple numerical scheme for implementing this algorithm is:

```
FOR each particle DO
  determine the 27 grid points
  add the contribution of the particle mass to the density
      at each of the 27 grid points.
END DO
```

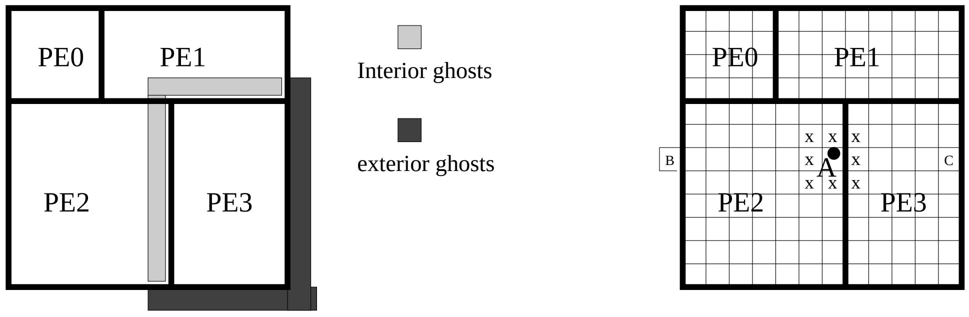
Assuming that the list of particles is distributed to the processors of the Cray T3D, then it is straightforward to port the simple numerical scheme to execute in parallel provided that *atomic update* primitives are used to ensure that when multiple processors write to the same PM grid point simultaneously, the writes will be serialized to yield correct results. If each processor is given $\frac{1}{p}$ of the total number of particles, then this simple scheme has the advantage that each processor will compute the effect of the same number of particles (perfect load balancing). However, this simple approach suffers from the following drawbacks.

1. Parallelism is enforced by using low level atomic operation. The more PEs used, the more memory conflicts will degrade the performance. Hence, the algorithm will not be scalable.
2. Each processor contains particles that may be scattered in the whole domain, which results in excessive interprocessor communication when updating the PM mesh.
3. The computation and communication are mixed, which degrades the processor utilization due to waiting for the communication to be finished.
4. Since the algorithm considers periodic boundary condition, the particles residing on the boundary of the domain must be treated differently when calculating their contribution to the PM grid points.

We may overcome these drawbacks by using Bisect to 1) separate the communication phase from the computation phase, 2) achieve load balancing and 3) treat all particles in the system in the same way. Specifically, Bisect is used to distribute the list of particles to the processors such that the particles in a given subdomain of the problem reside on the same processor (see Figure 3 (a)). The subdomains resulting from Bisect may be chosen to coincide with the uniform subdivisions of the PM mesh (Figure 2), or may be chosen such that each subdomain contains the same number of particles. In the former case, communication while interpolating from the particle masses to the PM mesh will be reduced, while in the latter case, the computation load will be balanced. The effect of both methods of applying Bisect will be studied later.

By grouping the particles that belong to each subdomain on the same processor, atomic updates are no longer needed when the effect of a particle mass is added to the density at a PM mesh point, as long as this mesh point is in the domain assigned to the processor. In fact, the use of the "domain extension" capability of Bisect may eliminate completely the need for atomic update operations. Specifically, each subdomain is expanded by one grid size (see Figure 3(a)) and the particles along the boundary of the grid are duplicated into the extended domain. The particles in the extended domain are called ghost particles, while the particles in the original domain are called real particles. With this technique, a particle on processor P will only contribute to the PM mesh points that belong to the subdomain assigned to processor P. Its contribution to the PM mesh points outside the subdomain assigned to P will be computed by the processor(s) that have a copy the particle as a ghost. For example, in Figure 3 (b), PE2 will compute the contribution of Particle A to the 6 grid cells in the domain that belongs to PE2. The contribution of A to the 3 grid cells in the domain that belongs to PE3 will be computed by PE3 using the ghost of A that exists on PE3. The density assignment algorithm is depicted in Figure 4.

Hence, the ghosting technique eliminates the interprocessor communication when the algorithm is executed in parallel provided that each processor has a disjoint domain and has all the particles within the extended domain. As shown in Figure 3, there are two types of ghost particles. Interior ghost particles are copies of real particles that lie within the domain,



(a) Domain decomposition with extension

(b) The effect of boundary particles on density

Figure 3: Particle to density interpolation in 2-D domains

```
for each particle in P (real or ghost) do
  determines the NGP (x, y, z)
  for i = x-1 to x+1 do
  for j = y-1 to y+1 do
  for k = z-1 to z+1 do
    if (i, j, k) is within the domain assigned to P then
      add the interpolated mass of the particle to mesh point (i, j, k)
```

Figure 4: Density assignment algorithm at a processor, P.

and exterior ghost particles are the particles duplicated in the entire problem domain. With periodic boundary condition, exterior ghost particles must be treated carefully. More specifically, the grid points that are out of the domain are considered to be the grid points in the wrap–around domain. An example is shown in Figure 3 (a), where the effect of the particles in cell B, which is a cell outside the domain, is equivalent to the effect of the particles in cell C.

Force field transformation

The second phase of PM is to calculate the force field from the density. Both the density and force fields are defined on the PM mesh. This phase is composed of a three dimensional Fast Fourier Transform (FFT), a multiplication by a specific function (a Green function) and an inverse 3D FFT operation. Since this phase does not involve particle operations, we will not discuss it in detail in this paper. Some optimization is done on the distribution of the data structures to reduce the total communication required by the FFT operations.

Extrapolating the force to the particles

This phase is the inverse of the density assignment phase. It also requires moving data between the two fundamental data structures, the PM mesh and the particle list. A simple numerical scheme for implementing this is:

```
FOR each particle DO
```

```
    determine the corresponding 27 grid points
    add the extrapolation of the forces on the grid points
        back to the particle
  END DO
```

This algorithm can easily be parallelized because each particle gets forces from the PM mesh — there are no collisions at the destination (the particle list). For this reason, it has long been recognized that it is much easier to vectorize this step compared to vectorizing the density assignment step [5].

When Bisect is used to distribute the particles uniformly among processors, the subdomain assigned to each processor may be different from the part of the PM mesh assigned to that processor (compare Figures 3 and 2). Hence, the interpolation/extrapolation steps between particles and PM mesh points will generate interprocessor communication. Such communication may be reduced if each processor caches the part of the PM mesh that corresponds to its subdomain, operates locally on this part of the mesh and then writes the result back into the global PM mesh.

The effectiveness of this caching scheme depends on the ratio of the size of the local PM mesh on a processor to the number of particles in that processor. If the ratio is high, caching may not be effective since it requires each processor to cache a large portion of the PM mesh which may not be updated, thus resulting in unnecessary communication. Moreover, the size of the local copy of the PM mesh will not be uniform across the processors, which creates problems regarding uniform memory allocation in processors. Anyhow, we have implemented the above caching scheme and found that its effect on performance is not significant.

5.2 Particle-particle method

The PP method calculates the short range forces between particles. The algorithm is divided into two phases. In the first phase, the particles are organized into cells. In the second phase, the PP force is calculated for each particle by checking all the particles in its cell and adjacent cells that are within the cut–off distance. By organizing the particles into cells, the complexity of the PP force calculation is reduced to $O(N \times\ maximum\ cellsize)$. The amount of computation for each particle is proportional to the number of particles that lie within the cut–off distance. The PP method can be easily parallelized if each processor contains a disjoint subdomain with appropriate ghost particles. In this case, a serial PP code is used on each processor to compute the PP forces on the particles in the subdomain assigned to the processor. Hence, particle reorganization, such as that done by Bisect, is essential for allowing the parallel execution of the PP method to proceed independently on each processor.

By assigning a weight to each particle equal to the total number of particles within the cut–off distance to the particles, Bisect is able to balance the workload for the PP force calculation. In addition, the domain extension with periodic boundary condition makes it possible to eliminate the ghosting phase in the PP force calculation. In each time step, the weight of a particle can be calculated while computing the PP forces. This weight can be used to balance the load in the next time step. Although particles move in each time step, the displacements are generally small, and the weights computed during a time step are, in general, good approximations of the workload for the particles in the next time step.

5.3 Performance

In this section we report performance results for the P3M program. We focus on comparing the effects of different load–balancing strategies. Table 8 shows the time for the PM calculation for 1 million uniformly distributed particles on a $64 \times 64 \times 64$ PM mesh. This table compares the running time for the PM routines with and without Bisect. In the table, the runtime for PM with Bisect includes the time to run Bisect, while the runtime without Bisect includes the time for atomic update operations. The speedup column records the speedup of the program running on PE processors over $PE/2$ processors. Although each processor has exactly the same number of particles for the PM routine without Bisect, the PM routine with Bisect has not only shorter execution time in all cases but also better scalability. This confirms that Bisect can achieve load balancing and reduce communication overhead.

Table 8: PM with and without Bisect

no. of PEs	PM without Bisect		PM with Bisect	
	execution time(sec)	speedup ratio	execution time(sec)	speedup ratio
16	11.02	–	8.66	–
32	6.05	1.82	4.61	1.88
64	3.60	1.68	2.48	1.86
128	2.86	1.26	1.48	1.68

Table 9 compares the different load balancing strategies on the PM and PP force computations. The most appropriate domain partitioning strategies for PM and PP are compared with the straight forward partitioning strategy in which the domain partitions coincide with the PM mesh partitions. That is a partitioning into subdomains of equal volumes, which is called partition *by domain* in Table 9. As discussed before, the work load for the PM force calculation is proportional to the number of particles in each processor, while the workload for the PP force calculation is proportional to the total amount of particle interaction in each processor. Hence, the most appropriate partitioning strategy for the PM phase is the one that balances the number of particles in each subdomain, while the most appropriate partitioning strategy for the PP phase is the one that balances the sum of the weights of the particles in each subdomain. The table shows the running times and the speedups gained by using Bisect to achieve load balancing. The results shown in the table are for 262144 (64^3) nonuniformly distributed particles with a PM mesh of size $64 \times 64 \times 64$. We can see from the table that proper load balancing speeds up both the PP and PM force calculation by a factor of more than 3 for all cases.

As discussed before, each of the PP and PM phases need to use different balancing criteria to achieve optimal load balancing. However, we want to move the particles among processors as little as possible to reduce the communication cost. Depending on the relative

Table 9: Performance of PP/PM with different load balancing criteria

no. of PEs	Particle–Mesh(PM)			Particle–Particle(PP)		
	by domain	by no. of particles	speedup	by domain	by weight	speedup
16	20.68	6.37	3.25	100.8	29.61	3.40
32	11.19	3.41	3.28	60.63	18.98	3.19
64	5.84	1.87	3.12	44.92	14.01	3.21
128	3.85	1.10	3.5	31.99	8.78	3.64

execution times of Bisect, PM and PP, the most efficient strategy for a P3M execution can be

1. Bisect to optimize PM; in this case, Bisect is called once in each iteration to perform load balancing for PM. PP will use the same particle distribution.
2. Bisect to optimize PP; in this case, Bisect is called once in each iteration to perform load balancing for PP. PM will use same particle distribution.
3. Bisect to optimize both PM and PP; in this case, Bisect is called twice in each iteration, once before the PM calculation and once before the PP calculation.

Table 10 shows the performance of these options for the P3M program. This experiment was done with 2 million non–uniformly distributed particles and a PM mesh size of $128 \times 128 \times 128$. We can see from the table that Bisect takes much less time compared to PM and PP. Therefore, it is more efficient to call Bisect twice in each iteration to balance both PM and PP. With two Bisect calls in each iteration, the running time for Bisect only accounts for 6.6% of the total P3M time in 16 PE system and 9.5% in a 128 PE system.

Table 10: Performance of P3M

	16 PEs			128 PEs		
	with Bisect for PM	with Bisect for PP	with two Bisects	with Bisect for PM	with Bisect for PP	with two Bisects
time for PM Bisect	3.3	0.0	3.3	1.2	0.0	1.2
time for PP Bisect	0.0	7.4	7.4	0.0	4.1	4.1
time for PM	25.4	62.9	25.4	6.4	18.2	6.4
time for PP	194.8	121.0	121.0	71.3	44.1	44.0
total	223.5	191.3	157.1	78.9	66.4	55.7

Another observation is that the runtime for Bisect is different for different load balancing criteria. For example, in a 128 PE system, Bisect takes 1.2 seconds to balance the number of particles while it takes 4.1 secods to balance the weights. The main reason for this difference is the load imbalance within Bisect. The load is balanced in Bisect when all the PEs have similar number of particles in each iteration. When the PEs have similar weights, they may have different number of particles, which causes load imbalance within the execution of Bisect. Nonetheless, the performance gain obtained by using Bisect always more than offsets the cost of invoking Bisect in our experiments.

6 Conclusion

Bisect is a powerful tool for domain decomposition on parallel systems. In addition to balancing the load in each subdomain, Bisect may augment each subdomain with copies of the objects it will need from other subdomains, thus reducing the communication overhead that may be needed for interaction on subdomain boundaries. From experimental data, we can see that the effectiveness of Bisect for balancing the load increases with the number of objects in the system. That is, Bisect is most effective when it is most needed. Moreover, by changing the number of bins in each direction, one can reach a compromise between the load balance achieved and the cost of achieving the load balance (the time to execute Bisect).

Parallelizing P3M program poses challenges in distributed memory systems. The Bisect package is able to perform load balancing and communication optimization for this program. In addition, with Bisect, we were able to separate the communication phase from the computation phase in the P3M program. Our performance evaluation shows that the cost of running Bisect is small relative to the gain obtained from balancing the load and it is more efficient to perform load balancing twice using Bisect in each time step in the P3M program.

References

[1] Balsara, D.S. and Brandt, A., "Multilevel Methods for Fast Solution of N-Body and Hybrid Systems", Int. Ser. Num. Math, 98, 131. 1991.

[2] Bokhari, S.H., "Assignment problems in Parallel and Distributed Computing", Kluwer Academic Publishers, 1981.

[3] Cybenko G., "Dynamic Load Balancing for Distributed Memory Multiprocessors", J. of Parallel and Distributed Computing, 7, 279-301, 1989.

[4] Ferrell, R. C. and Bertschinger E. "A Parallel Processing Algorithm for Computing Short–Range Particle Forces with Inhomogeneous Particle Distributions." In proceeding of the 1995 Society for Computer Simulation Multiconference, April 1995.

[5] Ferrell, R. C. and Bertschinger E. "Particle–Mesh Methods on the Connection Machine." In Int'l Journal of Modern Physics,

[6] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D. "Solving Problems on Concurrent Processors". Prentice-Hall, 1988.

[7] R.W.Hockney and J.W. Eastwood, "Computer Simulation Using Particles." New York: Mcgraw Hill International, 1981.

[8] Warren, M.S. and Salmon, J.K "A Parallel Hashed Oct-tree N-body Algorithm". In Supercomputing '93, PP. 12-21, 1993.

[9] X. Yuan, B. He, D. Balsara and R. Melhem, "A Load Balancing Package for Domain Decomposition in Distributed Memory System," International conference and Exhibition on High-Performance Computing and Networking(HPCN), LNCS 1067, pages 547-554, Brussel, Belgium, April 1996.