Parallelization of Particle Methods on the Sphere*

Ömer Eğecioğlu and *Ashok Srinivasan*

Department of Computer Science
University of California Santa Barbara, CA 93106
{omer, ashok}@cs.ucsb.edu

Abstract

We consider data structures and algorithms for efficient parallelization of particle methods when the domain is the surface of a sphere. Such applications typically arise when dealing with directional data. We propose a domain decomposition scheme based on geometric partitioning, that provides domains suitable for efficient implementation of requisite operations that are performed on the data in parallel. The algorithm has the advantage of being fast enough to be applied dynamically, and at the same time provides good partitions, comparable in quality to those produced by multilevel graph partitioning schemes.

1 Introduction

Particle methods are widely used in several applications [1, 5, 4, 3]. These typically involve a set of particles represented as points in some space, and a function that describes the interaction between pairs of particles. For each particle, one independently sums the interaction between it and all other particles, and a new state for each particle is then computed. This new state typically is a new position and velocity, and the calculations are repeated several times to observe the evolution of the system. This leads to an irregular computational problem in which the set of particles which interacts with any given particle changes with time in an unpredictable manner. Furthermore, in applications involving directional data such as complex fluid flow problems [21, 14], the natural domain of representation is the unit sphere.

In a parallel implementation, particles are assigned to processors by first breaking the domain into subdomains, and then mapping these subdomains to different processors. Each processor proceeds to compute the interactions of all the particles in the system with the particles in its subdomain. The interactions with particles that lie outside its subdomain require a processor to obtain the states of those particles from other processors. We call the particles in the subdomain of a processor the points *owned* by the processor. The points owned by a given processor that are needed by other processors are *shared* points. In many applications such as molecular dynamics and Smoothed Particle Hydrodynamics based methods, the interacting forces between the particles are short range and the effect of particles that are farther away than a certain cut-off distance can be ignored [4, 3]. In order to further mitigate high communication costs, one usually tries to overlap computation with communication. Hence, processors first send their shared data to the processors that need them, and following this perform their local computations. Subsequently, they receive the shared data they

*This technical report discusses parallelization of particle methods on the sphere. It consists of two parts (i) Domain decomposition, and (ii) data structures to facilitate such computations. The first part appears in the Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR'96), held in Santa Barbara, California, August 19-21, 1996. It is reproduced here for completeness.

1. Domain Decomposition.
2. Map domains to processors.
3. Start loop
   (a) Determine shared points.
   (b) Send shared points to processors that need them.
   (c) Compute using interior data.
   (d) Receive shared data needed from other processors.
   (e) Compute using received data and update state.
   (f) Update domain data.

   End loop

Figure 1: Outline of a general parallel particle method calculation.

themselves need and perform their remaining computations, and update the states of the points in their subdomain. Some updating of the domains can also be done at this stage. Usually a full domain decomposition is not performed at the end of each iteration since the cost of the decomposition can be prohibitive. The basic scheme of such calculations is outlined in Figure 1.

Efficient parallelization requires that we choose subdomains for each processor in such a way that only few particles outside the subdomain interact with particles within the subdomain. It is also necessary to efficiently determine the set of shared particles at each processor. An important aspect of the computations is to perform effective dynamic range searching so that one computes interactions with only those particles that are within the cut-off distance of the short range interactions.

The outline of this paper is as follows. Sect. 2 describes graph-theoretical and geometric domain decomposition strategies as they apply to particle methods on the sphere. The algorithm we present in Sect. 3 is essentially a geometric partitioning based on Orthogonal Recursive Bisection [2]. However, we take advantage of the geometry of the sphere to produce partitions with quality comparable to sophisticated methods such as spectral partitioning. Experimental results and comparisons with other popular schemes available in Chaco, version 2.0 [9] and Metis, version 2.0.3 [11] are presented in Sect. 4. These experiments show that our algorithm is an order of magnitude faster than even the relatively fast inertial method for large problem sizes, and demonstrate the high quality of the partitions obtained. In Section 5, we describe various sets and the operations that are performed on them, which abstract the essential features of the computations involved, and also describe our implementation strategy. We show how our domain decomposition strategy facilitates efficient implementation of other operations on the data. Conclusions are given in Sect. 6.

2 Domain Decomposition

Domain decomposition has been widely studied [9, 10, 11, 12, 13] and several types of methods for its solution have been proposed: graph-theoretical and geometric, for example. Graph-theoretical schemes ignore coordinate information and treat domain decomposition as a general graph partitioning problem. Geometric algorithms, in contrast, use coordinate information of the points to divide the domain into contiguous regions.

The quality of the partitions produced can be judged by the load imbalance introduced and the communication cost incurred. We try to keep the load balanced, i.e., ensure that the amount of computation performed by each processor is about equal. Subject to this restriction, we further wish to keep the communication cost low. There are several measures for estimating the communication cost. Before describing the criterion used in our algorithm, we shall briefly describe the communication pattern implied by Figure 1. In order to overlap computation and communication, each processor determines the shared data it needs to send to other processors. It is appropriate to send the required data to each processor as a single message. This reduces the startup time and is especially advantageous in systems that support long messages. If n_i is the number of processors that need the data of point i, excluding the processor to which i has been assigned, then we define the communication cost as $\sum n_i$. This measure ignores the startup time and can be justified if the messages are sufficiently long. It differs from the *hop* metric in that the number of links traversed are ignored, providing an architecture independent measure. With cut-through routing being widely prevalent, this criteria seems justified. However, it should be noted that too many messages in the system could cause network congestion and the number of links traversed could affect the true communication cost [9]. Our communication measure also differs from the commonly used edge-cut metric in graph partitioning which tries to minimize the number of edges cut. If more than one point in a particular processor needs data related to one point from another processor, the remote processor need send the data only once. The processor that receives the data can store this and reuse it when needed. In contrast to the edge-cut metric, our communication cost takes this factor into account as well.

Graph-theoretical algorithms such as spectral methods produce high quality partitions especially when combined with a local refinement strategy [13], but require too much time. When combined with multilevel methods, these give good partitions much faster [10], however, they are still not fast enough to be used frequently. Since the distribution of the points could change significantly in the types of application we are considering, the quality of the partitions may degrade quickly.

Geometric algorithms make use of the coordinates of the points to find partitions fast. In this case the quality of the partitions obtained is usually not very good. Orthogonal Recursive Bisection (ORB) for example, bisects the domain along a coordinate. This is recursively applied using different coordinates. This scheme is fast, though the quality, as judged by the communication cost incurred, could be poor. Another method that uses coordinate information is the Inertial Method. This method produces partitions which are usually of a higher quality than those produced by ORB, at the expense of a slight increase in the computational effort required to produce the partitions. Alternate approaches to parallelization of particle methods can be found in [15]. Our scheme resembles ORB, but takes advantage of certain metric properties of the surface of the sphere to give good partitions.

Before describing our algorithm, we note that one can consider stereographically projecting the points on the sphere onto the plane and then using an existing partitioning algorithm for the plane. However, points close to the projecting pole are widely separated in the projection. This distortion on locality makes geometric algorithms unsuitable.

3 The Algorithm

We propose a combination of a geometric and a combinatorial scheme for decomposing the domain into regions bounded by a pair of latitudes and and a pair of longitudes. The input to the domain decomposition algorithm consists of: (i) the number P of processors available, (ii) the cut-off distance h for the short-range interactions, and (iii) a set of N points defined by their coordinates (latitude and longitude) on the unit sphere. We assume that each point is dynamically assigned a positive weight which is proportional to the computational effort required for computing the new state of the point. Each point represents a particle in the system.

The computational effort for a given point is roughly proportional to the number of points within a distance h of the given point. There are different options available to obtain a reasonable estimate of what the weight should be. A large class of problems involves compressible fluid flow calculations in which the density of the fluid has to be determined [4]. For sufficiently small h, the number of points within a distance h of any point is approximately proportional to the density at the point. Thus, one may take the weight to be proportional to the density. If such data is not available, one may use non-parametric density estimation techniques to estimate the density [6, 7]. In our implementation of the domain decomposition algorithm on the sphere we have used positive integral weights, though using floating point weights does not present any additional difficulty.

As its output, the domain decomposition algorithm associates an integer in $[0, P-1]$ with each point, which indicates which partition (subdomain) a point belongs to. In addition, the algorithm can also produce a mapping of partitions to processors, thus determining the processor to which each point is assigned. Our algorithm produces a mapping for a tree topology, with the processors located at the leaves of the tree. This is not unduly restrictive since efficient schemes exist for embedding trees into other topologies [8].

We use a recursive bisection procedure. At each stage, we first consider a cut on the subregion along the latitude that gives a balanced load, and also a longitudinal cut that gives a balanced load. We then calculate and compare the communication cost of each cut, and choose the one that with the lower cost. Thus initially, one of the first two types of cuts shown in Figure 2 is made. Being on the

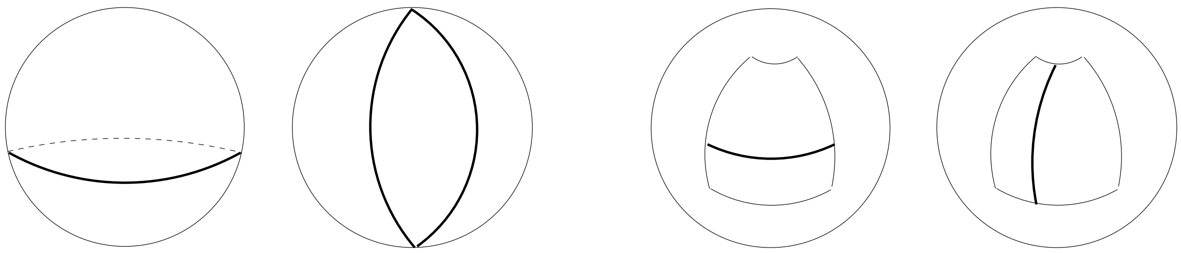

Figure 2: Initial latitudinal and longitudinal cuts; generic latitudinal and longitudinal cuts.

surface of the sphere adds an additional complication if the region we are considering has not been cut by a longitude before. Such regions look either like rings or polar caps, as shown in Figure 3. In this case there is no unique pair of longitudes that balances the load. In fact if there are n points in such a region, then there are $O(n^2)$ possible pairs of longitudes, and $O(n)$ possible pairs among these along which we can perform a cut while keeping the load balanced. A naive strategy would determine for each fixed choice of the starting longitude of the region, the corresponding longitude to end the region which gives a balanced load. This can be done by using a sorted array of longitudes in $O(\log n)$ time per initial choice. Since there are $O(n)$ possible initial choices, the overall time would be $O(n \log n)$ for the determination of the cut. However, we show that this type of a longitudinal cut can actually be determined using only $O(n)$ operations. For our algorithm we require an initial preprocessing overhead of $O(N \log N)$. However $O(N \log N)$ preprocessing is required for sorting the N points in the system according to a number of parameters anyway, so this does not add to the overall complexity. In this way, the rest of the computation takes only linear time at each level of the recursion.

The algorithm is outlined in greater detail below. Being on the surface of a sphere leads to some complicated boundary conditions to keep track of. In order to ensure the clarity of the presentation, these are not elaborated on here.

3.1 Latitudinal cut

We first describe latitudinal cuts since they are the simpler of the two. For a latitudinal cut, we wish to divide the sphere into two parts along a certain latitude. If we sweep a latitude from the South to the North Pole, then the load balance of the two parts it divides changes discontinuously, and does so exactly at those latitudes at which the data points are located. Hence, we need to consider only these latitudes. In the preprocessing step, we keep a list of the points sorted in ascending order of latitude. We traverse this array, and perform a prefix computation to calculate the cumulative weights of all those points that have been encountered so far, including the current location. In order to find the latitude at which to cut, we perform a binary search on the array of cumulative weights to find the location that gives the best load balance, i.e., has cumulative weight closest to half the total weight of the partition. It is possible for two adjacent locations to have equally good load balance. In this case we choose one of these cuts arbitrarily. We observe that if several points have the same latitude, they may fall in different partitions. In order to compute the cost of the chosen partition, we need to know the number of points that lie within a distance h of the closest latitudes of either region. We use the property that the shortest distance between a point and a latitude is the absolute value of their differences in latitude. We locate the farthest points within the cut-off distance of the latitudes defining the boundaries of the two regions. This is performed by binary search on the sorted array of latitudes. The number of points that will be communicated is one more than the difference in the positions of these points in the sorted list.

3.2 Longitudinal cut

There are two cases to consider for a longitudinal cut. If any ancestor (in the recursion tree) of the region being cut has been subjected to a longitudinal cut before, then this case leads to a simpler algorithm since cutting along a single longitude will divide the region into two parts. We choose this longitude in a manner analogous to that of the latitude, using a pre-sorted array of longitudes. Computing the cost requires a little more attention since the shortest distance between a point and a longitude is in general not the difference of their longitudes. If we sweep a longitude around the sphere and consider the neighborhood within distance h of this longitude, then points may enter and leave this neighborhood at different values of the longitude. We can calculate the longitudes at which any point enters and leaves this neighborhood. If a point is within distance h of all longitudes (near the poles), then we consider it as entering at $-\infty$ and leaving at ∞. We have a preprocessing step in which we create two sorted arrays based on the longitudes at which a point enters and leaves the neighborhood respectively. In order to compute the number of points within distance h of a given longitude, we find the last point that has just joined the neighborhood and the next point that leaves the neighborhood. These points are found by binary search on the two sorted arrays constructed above. The difference between the number of points that have joined the neighborhood and the number that have already left gives the number of points still in the neighborhood.

In the second type of longitudinal cut, no ancestor of the region has been subjected to a longitudinal cut before. Now we need to decide on two longitudes at which to perform the cut, since one longitude alone will not divide the region into two pieces (see Figure 3). We start with the first longitude, say A_1, in the sorted list of longitudes, and find the corresponding longitude A_2 that balances the load if we start the region defined by the cut at A_1. We compute the cost as in the case of a cut which requires a single longitude; however, we now need to add the number of points on both boundaries. We next look at the longitude B_1 that comes second in the sorted list of longitudes. We find the other end B_2 of the possible cut not by performing a binary search on the list, but

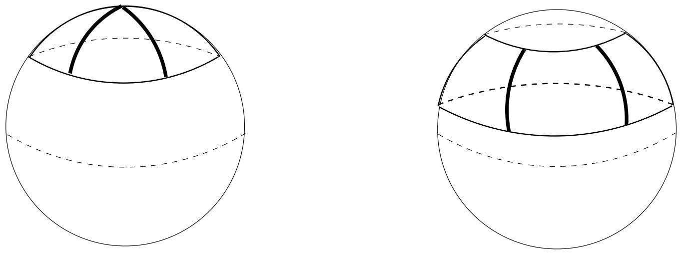

Figure 3: Longitudinal cuts that require a pair of longitudes.

by incrementing the location of A_2 until we achieve load balance. This construction is as shown in Figure 4. A similar procedure is carried out in computing the communication cost as well. We

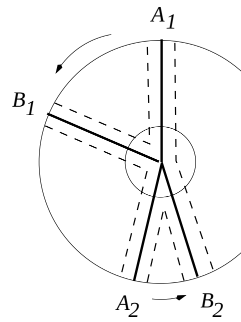

Figure 4: Pairs of longitudes that balance the load. The center is the North Pole.

proceed in this manner around the sphere and choose the cut that has the lowest communication cost.

An example of a sequence of various latitudinal and longitudinal cuts made using this decomposition scheme is given in Figure 5.

3.3 Partitioning

After we decide on a particular cut, we need to partition the set of points. An important step here is to obtain the sorted arrays for each partition without needing to perform an $O(n \log n)$ sorting step. We can get the sorted arrays in linear time by scanning each already sorted array in order and placing a point at the end of the array of whichever partition it belongs too. In addition, if we performed the longitudinal cut involving two longitudes, we need to change the values of the longitudes of certain points in one of the partitions by subtracting 2π from their coordinates so that if we sweep a longitude from one end of the partition to the other, we will remain within the partition. Getting the sorted arrays in this case also require two passes over the sorted arrays, rather than one as in the other cases.

3.4 Complexity

The complexity of the initial preprocessing step is $O(N \log N)$ required for sorting the N points. If there are n points in a partition that is to be further partitioned, then locating the latitude, or the longitude in a longitudinal cut involving only one longitude, can be done in $O(\log n)$ time. This takes $O(n)$ time for the longitudinal cut that requires two longitudes, since the "arms" A_1, A_2 and B_1, B_2 of Figure 4 sweep through the points once, without any backtracking. Generating the new partition

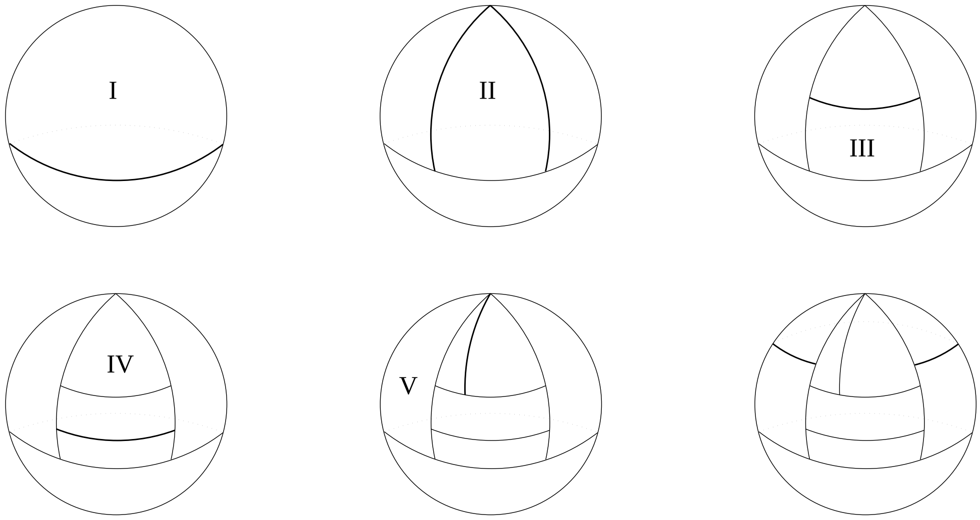

Figure 5: An example: The first cut is latitudinal, producing domain I and its complement. Sample subsequent cuts on some of the subdomains are: longitudinal cut on I, latitudinal cuts on II and III, longitudinal cut on IV, and latitudinal cut on V.

requires $O(n)$ time for any of these types of cuts, since we need to produce the sorted arrays too. Thus each level of the recursion requires $O(N)$ time leading to $N \log P$ complexity for the recursion if P parts are needed (assuming P is a power of 2). In addition, we have $O(N \log N)$ preprocessing time, which gives a total complexity of $O(N \log N + N \log P) = O(N \log N)$.

4 Experimental Results

We performed experiments to test the performance of our algorithm, and also to compare it with existing algorithms. As a measure of the load imbalance, we considered the quantity: tP/T where P is the number of processors desired, T is the sum of the weights of all the points, and t is the sum of the weights of all the points in the processor with the largest such sum. If the load were perfectly balanced, then this quantity would be 1. In all of the experiments, we found that all the methods tested gave well balanced partitions, especially with a large number of particles. Thus we do not report further on this aspect of the experiments, and judge the quality of the partitioning based only on the communication cost incurred. In the rest of this section, the term *quality* refers to the communication cost incurred. The ratio of the communication costs that appear in our plots to $P \cdot N$ is the average communication cost incurred between pairs of processors, as a fraction of the total number of points. We obtained our data points on the sphere by generating samples from three different probability density distributions. We used the rejection-acceptance technique to generate the points unless otherwise stated. We assigned weights to each point proportional to the density of the distribution at the point. The weights were rounded to integers with a minimum value of 1. We chose a value of h such that it gave a reasonably good estimate when using kernels for non-parametric estimation of the probability density. This was done on the basis of practical applications in which our scheme is particularly useful [14].

We compared our algorithm with general graph partitioning algorithms, since these have been found to give good quality partitions [10]. We also compared our scheme with the Inertial Method, since this is a geometric method which is much faster than the general graph partitioning methods. For problems of large size, even the multilevel graph partitioning algorithms were at least two orders of magnitude slower than our algorithm. Therefore we have presented timing results comparing our algorithm only with the inertial method. The inertial method used was that implemented by Chaco, version 2.0. The spectral bisection method used the multilevel spectral eigensolver implemented in Chaco, version 2.0. The multilevel spectral bisection scheme implemented in Metis, version 2.0.3 gave partitions of similar quality, and requiring a similar run time, as that of Chaco. However, the latter was marginally faster and gave partitions of slightly better quality in a few tests, and so was used in our experiments. We also tested the graph growing partition implemented in Metis for one test case. It should be noted that though the number of vertices in our graphs is not very high compared with many of the graphs used for tests in current literature, the relative denseness of our graphs results in a large number of edges. Many of our larger graphs have millions of edges.

Experiment 1:
We first performed experiments to observe how the speed of our method scales with the number of points. The experiments were performed on Sun SPARCstation 5, and the timing results are reported in seconds. We present results showing the total time required, and also the time required by just the partitioning phase, ignoring the initial preprocessing time. We compare it in Figure 6 with the inertial method implemented in Chaco, version 2.0, without the Kernighan-Lin refinement. The time reported is that taken just for the partitioning. We did not use the Kernighan-Lin refinement for the inertial method because this increased the time taken significantly. Since speed is the major advantage of a geometric algorithm, we decided not to use this refinement. It can be seen that as the number of points increases, the inertial method is about an order of magnitude slower than our scheme. We also note that our scheme scales well as the number of points increases. It should also be noted that with a large number of points, the majority of the time is spent in the initial sorting, which can be easily parallelized.

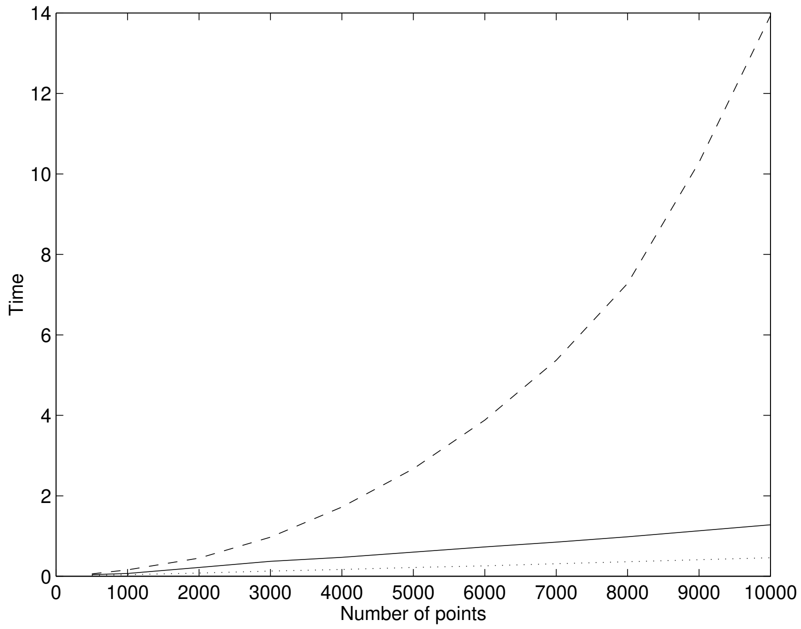

Figure 6: Comparison of the speed of our algorithm and inertial partitioning into $P = 32$ parts, without Kernighan-Lin refinement. The solid line shows the time taken by our algorithm. The dotted line shows the time taken in just the partitioning phase. The dashed line shows the results for the inertial method. Time is in seconds.

Experiment 2:
We next investigated the quality of the partitions obtained. In all the experiments reported below, the data was partitioned into $P = 32$ parts. We consider points generated on the unit sphere with the distribution: $\psi(\phi, \theta) = \exp(U \sin^2 \phi)/A$, where U is a parameter, A normalizes ψ to a probability density, and ϕ and θ are the latitude and the longitude respectively. This particular function arises in the solution of a certain problem in complex fluids [14], and the distribution depends only on the latitude. Here, $-\pi/2 \leq latitude \leq \pi/2$, and $0 \leq longitude < 2\pi$. In our experiments, we have used the special case of $U = 4.6$, $A = 25.6$. The value of h was taken to be 0.2.

We can see from Figure 7 that our method, inertial partitioning, and the multilevel spectral method give partitions of comparable quality for this distribution, though the spectral method gives slightly better partitions.

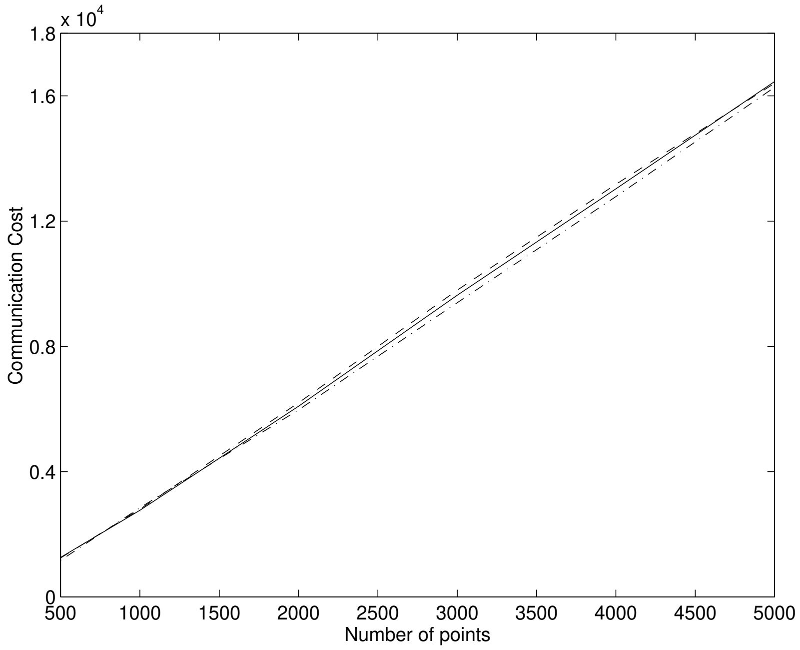

Figure 7: Comparison of the quality of the partitions, as judged by the communication cost incurred for the ψ distribution. The solid line presents the results for our algorithm. The dashed line shows the same for the inertial method. The dashed-dotted line shows the results for the multilevel spectral method with Kernighan-Lin refinement.

Experiment 3:
We next consider the distribution proportional to $1/\cos\phi$. This is obtained by choosing samples in which the latitudes and longitudes are uniformly distributed. This too gives a distribution which is independent of the longitude; however, the weights of the points vary greatly depending on the location. Thus, this is a good test of the algorithms with widely varying weights. The value of h was taken to be 0.3. Our method appears to do marginally better than the spectral method, and moderately better than the other methods. The results are presented in Figure 8.

Experiment 4:
Our final comparison is with the distribution given by: $\cos(\phi) \cdot \beta(\theta, 6, 2)$ where the β distribution is similar to the *beta* distribution, with the range scaled to $[0, 2\pi]$. The value of h was taken to be 0.3. This is non-uniform in both the latitude and in the longitude. It should be noted that most of the points are concentrated near the equator of the sphere due to the $\cos(\phi)$ term. Hence, we can expect

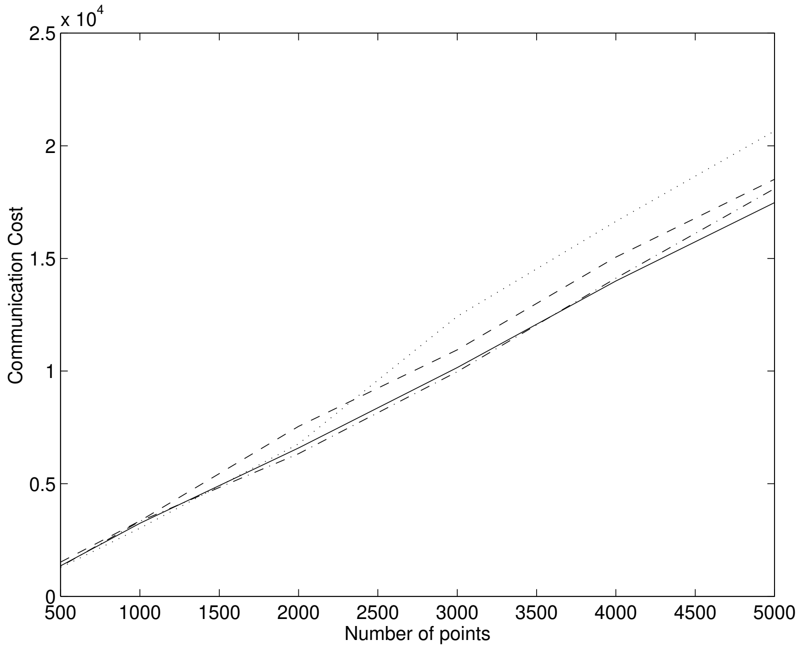

Figure 8: Comparison of the quality of the partitions, as judged by the communication cost incurred, for the distribution proportional to $1/\cos\phi$. The solid line presents the results for our algorithm. The dashed line shows the same for the inertial method. The dashed-dotted line shows the results for the multilevel spectral method with Kernighan-Lin refinement. The dotted line presents the results for the graph growing partition with boundary Kernighan-Lin refinement.

that the latitude cuts in our algorithm will not be particularly effective and most of the cuts will need to be longitudinal. Thus, this tests our algorithm under situations in which it is likely to be less effective. However, the results presented in Figure 9 show that our algorithm performs almost as well as the other algorithms in this case, though it is slightly worse than the multilevel spectral method.

The experiments conducted suggest that all the methods considered give partitions with good load balance. It also appears that the communication cost incurred are comparable. However, the scheme we have presented is much faster. This is probably due to the fact that this method has been specifically designed for points on the surface of a sphere, whereas the other methods are more general. We shall also see later that this method produces partitions such that other operations on the data can be implemented by fast algorithms.

5 Abstract Data Types and Implementation

We wish to parallelize computations which involve short-range interactions, with a cut-off distance h, between pairs of particles. We shall assume that we use P processors. Each processor is assigned all the points belonging to a connected region in space. The points owned by a processor can be considered of two types: (i) internal, and (ii) shared. The internal points are not needed for the computations of any other processor. The shared points are needed by at least one other processor. We consider as our domain the unit sphere, but the general framework given here is applicable to other domains as well.

Each particle in the system is represented by a point on the unit sphere, and will be defined by three fields: a latitude, a longitude, and a weight. The weight is proportional to the computational

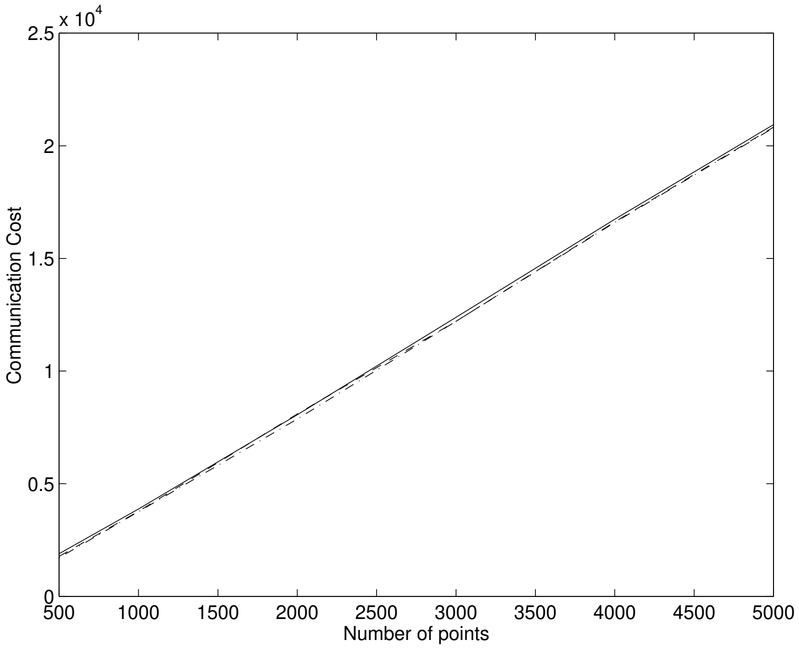

Figure 9: Comparison of the quality of the partitions, as judged by the communication cost incurred, for the distribution proportional to $\sin(\phi) \cdot \beta(\theta, 6, 2)$. The solid line presents the results for our algorithm. The dashed line shows the same for the inertial method. The dashed-dotted line shows the results for the multilevel spectral method with Kernighan-Lin refinement.

effort involved in calculations of the necessary fields at that point.

5.1 Abstract Data Types

Based on the type of data, we can define the following sets:

- U: The universe, consists of all the points in the system.
- O_i: The set of points owned by processor i. Processor i is responsible for computing the state for the points in O_i.
- N_i: The set of points not in O_i that are needed for the computations at processor i.

Described below are the operations that need to be performed on each of these sets. The Steps referred to appear in Figure 1:

U1: $Partition(U, P)$ partitions the universe into P subsets, O_i, $0 \leq i < P$ such that the sum of the weights of the points in each O_i are approximately equal. We would like the communication cost to also be low. This corresponds to Step 1. Step 2 is also usually related to this step.

O1: $Intersection(O_i, N_j)$ for each j at processor i. This determines the points owned by i that are needed by processor j. These points are sent to processor j. This corresponds to Step 3a.

O2: $RangeSearch(O_i, pt)$ returns the set of points in O_i that are within the cut-off distance h of point pt, for each point in O_i. This is used in Step 3c.

O3: $UpdateDomain(O_i)$ updates the set O_i after the new positions of all the points have been computed. This operation involves insertion/delete operations on O_i and corresponds to Step 3f.

O4: $ExtractN(U, O_i)$ determines the new set N_i consisting of external points needed by processor i. This is needed for Step 3a.

N1: $Intersection(O_i, N_j)$ for each j at processor i. This determines the points owned by i that are needed by processor j. These points are sent to processor j. This corresponds to Step 3a.

N2: $RangeSearch(pt, O_i)$ returns, for each point pt in $Intersection(O_j, N_i)$ received from processor j, the set of points in O_i within its cut-off distance.

It should be noted that the sets returned by *Intersection*, *RangeSearch*, and *ExtractN* need not be exact. They can return a superset of the true set. However, we would like this superset to be as close to the true set as possible, with the proviso that the implementation stays reasonably efficient.

5.2 Implementation

We wish to use the structure of the calculations, and the geometry of the surface, to devise an efficient implementation. In the algorithm we have devised, the shape of the domains is restricted to regions bounded by two latitudes and two longitudes. This enables efficient implementations of many of the operations required. The data structures associated with each set described in 5.1 are given in Figure 10. Here *latitude* and *longitude* are the coordinates of the points on the unit sphere; *weight*

- *Point*
  {
  float latitude, $0 \leq latitude \leq \pi$;
  float longitude, $0 \leq latitude < 2\pi$;
  integer weight $weight, > 0$;
  }

- O_i
  {
  linked_list sorted_latitude;
  linked_list sorted_longitude;
  Point points[n_i];
  RangeTree tree;
  }

- U
  {
  Point points[N];
  }

- N_i
  {
  float lat_{min};
  float lat_{max};
  float $long_{min}$;
  float $long_{max}$;
  }

Figure 10: Data structures.

is a measure of the amount of work associated with computing the state of the point; *sorted_latitude* is a doubly linked list with pointers to both ends that lists the points in ascending order of their latitude value. *sorted_longitude* is a similar list that gives the points in ascending order of their longitudes. *points* is an array consisting of the points in the given set. *tree* is a range tree used for range searching. If a simpler range searching is to be performed, then this field is not used. lat_{min}, lat_{max}, $long_{min}$, and $long_{max}$ are the minimum and maximum latitude and longitude respectively of all the points in the domain belonging to the processor.

We describe below our implementation of the various operations performed on the sets mentioned so far, except the partitioning phase which has already been described.

5.2.1 ExtractN

We need to determine N_i, that is, the set of points external to processor i that are needed for the computations at processor i. If we were to actually determine each point in the set, this would result in a high communication cost. Since we have defined the domain of any processor to be regions bounded by two latitudes and two longitudes, we can characterize the set N_i without actually constructing it. The set N_i is a subset of all those points that lie in a region within distance h of the boundary of the domain of processor i. This region can be determined once we know the minimum and maximum latitudes and longitudes of the region. These values are obtained as a result of our domain decomposition algorithm. Thus, this operation can be performed in constant time. It should be noted that this characterization of N_i gives a superset of the true set. However, we permit this, as mentioned earlier. If we have a large number of points, then the looseness of this characterization can be considered the difference between the convex hull of the set of points in processor i and the boundary of the domain as characterized above. If the points come from a continuous distribution, as is normally the case, then this difference diminishes as the number of points increases. Thus, this is a reasonable characterization of the set.

5.2.2 Intersection

If the exact result is needed for $O_i \bigcap N_j$, then the communication cost will be high due to the cost of sending the set N_j. Since we have characterized N_j with four parameters, the communication cost is actually low. In practice, we can store these coordinates at each processor that needs these values after the domain decomposition. Thus, no additional communication cost is incurred. These values will need to be changed only each time we perform a full domain decomposition. If processors i and j are separated by a latitude, we can scan the list of sorted latitudes to find the necessary points to send. A similar strategy is used for the longitudes. If there are K points within distance h of the boundary, then this operation can be performed in $O(K)$ time. It should be noted that as an efficient implementation of this operation, we do not find intersection for those sets for which the result will be the empty set.

5.2.3 UpdateDomain

As a result of the change in position of the points at the end of each iteration, points in some processors will move to the domains of a neighboring processor. These points will have to be deleted from the current processor and inserted into the other processor, after a communication step. We can retain the same domains for a set of iterations, until it is determined that the current domains are no longer of sufficiently good quality. We then perform a domain decomposition again and repeat the same process. If K points are inserted or deleted from a region containing n points, then we can perform these operation on the linked lists in time $O(Kn)$. Since we expect $K \ll n$, this operation is fast.

5.2.4 Range Search

Several efficient algorithms have been proposed for range-searching in Euclidean space. We can use the property that we are allowed to get a superset of the true set to adapt these algorithms for the sphere. Given any point, and h, we can determine bounds for the latitudes and longitudes within which all the desired points lie. If we create a data structure for range searching using the latitude and longitude as coordinates in 2-D, we can determine the desired point by performing a range search at most twice. We require two searches if the required region spans the 0 longitude. Range search in 2-D can be done in time $O(n \log n + K)$ time using the Willard-Lueker modification of the range tree [16]. Since we are required to perform updates such as insertion and deletion, we can use any of the dynamic update methods reviewed in [17]. Algorithms also exist for searching on a disc [18]. However, often simpler procedures are sufficient. For example, a simple range searching strategy that fits in with the rest of the computations, is to project the data onto one dimension [19]. Then we can sort this projected data and perform a range search based on the projected coordinates. Due to the triangle inequality, this will retrieve a superset of the true set. We can use the sorted lists we use for the other operations, and which can be obtained from the domain decomposition algorithm, to perform this operation.

5.3 Source Code

The source code of the implementation can be obtained from the authors if further details of the implementation are required.

6 Conclusions

We have presented a geometric domain decomposition algorithm that partitions data on the surface of a sphere and gives partitions suitable for particle method applications. Experiments have shown that the quality of the partitions obtained are comparable to more sophisticated schemes. This method has the advantage of being extremely fast, even compared with the inertial method. This good performance is to be expected since the graph edges in our problem are a function of geometric locality, and so one would expect a geometric algorithm to do well. Another advantage of geometric algorithms for this problem is that the graphs have a lot of edges, and so these algorithms avoid the high storage costs associated with graph algorithms. Another advantage of this method is that the domains produced can be parameterized well using four coordinates. We show that this property is useful for other operations. We have defined the sets and operations that need to be performed in the parallel algorithm, and presented efficient implementation of these algorithms. The distinguishing feature of this work is that we take a unified approach to parallelization. Thus, our domain decomposition is suited to the task of reducing communication overhead, and of efficient implementation of the operations that need to be performed.

References

[1] Barnes, J., Hut, P.: A hierarchical $O(N \log N)$ force-calculation algorithm. Nature **3** (1986) 446-449

[2] Berger, M.J., Bokhari, S.H.: A partitioning strategy for nonuniform problems on multiprocessors. IEEE Transactions on Computers **C-36** (1987) 570–80

[3] Young, W.S., Brooks III, C.L.: Implementation of a Data Parallel, Logical Domain Decomposition Method for Interparticle Interaction in Molecular Dynamics of Structured Molecular Fluids. Journal of Computational Chemistry **15** (1994) 44-53

[4] Monaghan, J.J.: Particle Methods for Hydrodynamics. Computer Physics Reports **3** (1985) 71-124

[5] Harlow, F.H.: The Particle-in-Cell Computing Method for Fluid Dynamics. Meth. Comput. Phys. **3** (1964) 319-343

[6] Eğecioğlu Ö., Srinivasan, A.: Efficient Nonparametric Estimation of Probability Density Functions. Technical Report TRCS95-21, University of California at Santa Barbara, 1995

[7] Eğecioğlu Ö., Srinivasan, A.: A Fast Nonparametric Density Estimation Algorithm. Communications in Numerical Methods in Engineering (In Review)

[8] Leighton, F.T.: Introduction to parallel algorithms and architectures : arrays, trees, hypercubes. M. Kaufmann Publishers, San Mateo, California, 1992

[9] Hendrickson, B., Leland, R.: The Chaco User's Guide, Version 2.0. SAND95-2344, Sandia National Laboratories

[10] Hendrickson, B., Leland, R.: A Multilevel Algorithm for Partitioning Graphs. SAND93-1301, Sandia National Laboratories

[11] Karypis, G., Kumar, V.: METIS, Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0. Dept. of Computer Science, University of Minnesota, 1995

[12] Williams, R.D.: Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations. Concurrency **3** (1991) 457-481

[13] Kernighan, B.W., Lin S.: An Efficient Heuristic Procedure for Partitioning Graphs. Bell System Technical Journal, 1970

[14] Chaubal, C., Leal, L.G.: The Effect of Flow Type on the Rheology of Liquid Crystalline Polymers. Society of Rheology, 67th Annual Meeting, Sacramento, CA Oct 8-12, 1995

[15] Plimpton, S.: Fast Parallel Algorithms for Short-Range Molecular Dynamics. J. Comp. Phys. **117** (1995) 1–19

[16] Preparata, F.P., Shamos, M.I.: Computational Geometry. Springer-Verlag, New York, 1985

[17] Chiang, Y., Tamassia, R.: Dynamic Algorithms in Computational Geometry. Proceedings of the IEEE **80** (1992) 1412-1434

[18] Chazelle, B.M., Edelsbrunner, H.: Optimal Solutions for a Class of Point Retrieval Problems. Tech. Rep. CS-84-16, Dept. of Computer Science, Brown University, 1984

[19] Friedman, J.H., Baskett, F., Shustek, L.J.: An Algorithm for Finding Nearest Neighbors. IEEE Transactions on Computers **C-24** (1975) 1000-1006

[20] Barnard, S.T., Simon, H.: A parallel implementation of multilevel recursive spectral bisection for application to adaptive unstructured meshes. In: Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, CA, USA, 15-17 Feb. 1995. Edited by: Bailey, D.H.; Bjorstad, P.E.; Gilbert, J.R.; Mascagni, M.V.; and others. Philadelphia, PA: SIAM, 1995, 627–32

[21] Szeri, A., Leal, L.G.: A new computational method for the solution of flow problems of microstructured fluids. Part 2. Inhomogeneous shear flow of a suspension. Journal of Fluid Mechanics **262** (1994) 171–204