WILLIAM ERIK SHEPARD

A Parallel Approach to Searching for Nearest Neighbors With Minimal Interprocess Communication

(Under the Direction of E. LYNN USERY)

This research exploited a classic sequential method for nearest neighbor searching in order to develop a parallel search approach. This approach was implemented in an inverse distance weighting routine in order to demonstrate efficacy. A set of 100 sample point datasets was developed and interpolated using the sequential implementation of inverse distance weighting present in ARC/INFO and the parallel implementation developed in the course of this work. Run times were captured for each run in both sets of 100 and compared to absolutely and proportionally in order to assess improvement using the parallel methodology over the sequential methodology. Experimental results showed that, while sequential processing times increase exponentially, parallel processing times increase linearly. The mean proportion of parallel time to sequential time for the 100 runs was 17% with a 5% standard deviation (maximum of 27% and minimum of 8%), increasing exponentially as input and output sizes increased linearly.

INDEX WORDS: GIS, Parallel processing, Nearest neighbor search, Inverse distance weighting, Kriging, Projection, Geostatistics, Geoprocessing

A PARALLEL APPROACH TO SEARCHING FOR NEAREST NEIGHBORS WITH MINIMAL INTERPROCESS COMMUNICATION

by

WILLIAM ERIK SHEPARD

B.S., The University of Georgia, 1995

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA
2000

A PARALLEL APPROACH TO SEARCHING FOR NEAREST NEIGHBORS WITH MINIMAL INTERPROCESS COMMUNICATION

by

WILLIAM ERIK SHEPARD

Approved:

Major Professor: Dr. E. Lynn Usery

Committee: Dr. Thomas W. Hodler
Dr. Suchendra Bhandarkar

Electron Version Approved:

Gordhan L. Patel
Dean of the Graduate School
The University of Georgia
August 2000

ACKNOWLEDGEMENTS

The writer wishes to thank Dr. E. Lynn Usery, The University of Georgia for his advice and support in completing this thesis; Dr. Thomas Hodler, Associate Dean of the Graduate School, The University of Georgia, and Dr. Suchendra Bhandarkar, Department of Computer Science, The University of Georgia, for serving on the committee and for contributing invaluable help each in their respective areas. The writer also wishes to thank The Office of Information Technology Outreach Services, The University of Georgia, for providing computer facilities to complete this research. Finally, the writer wishes to thank his wife, Julie, for her unwavering support during this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Hardware limitations due to processing power and memory constraints have been among the principal difficulties that prohibit geostatistical analysis with increasingly large datasets and increasingly fine resolutions. Although the situation is improving as processor speeds rise and chip and memory costs fall, there is a finite upper bound to the degree of improvements which may be achieved through such hardware advances. In order to increase processor power, more and more components must be squeezed into a limited area on the chip. This in turn requires ever shortening pathways between components. However, as pathway lengths shorten, the separation between components also decreases – leading to interaction between the components in some interesting, nonlinear ways. This interaction places a limit on the lower pathway length bound and thus on the maximum chip speed (at least in current architectures). There are also limitations imposed in the arena of available memory which are not currently physical, but are rather procedural. Current operating systems only provide support for a finite amount of random access memory. This limits the volume of data which may be accessed by the processor at any one time. Using disk read and write operations to swap a subset of data in and out of memory can alleviate this limitation at the expense of performance. Although disk seek times have improved dramatically, they are nevertheless far inferior to random access memory seek times (by three orders of magnitude).

Significant research is underway by organizations such as Cray, IBM and Intel to develop and extend parallel processing capabilities where multiple processors

operate simultaneously on either the complete dataset or a subset of it. These efforts, however, tend to focus on massively parallel problems – computing galactic positions, nuclear reactions, etc. – on thousands of processors. Such efforts are prohibitively priced for most enterprises and are also excessive. Problems such as geostatistical analysis may benefit from parallel processing approaches, but may not require the power of such massively parallel systems. A more cost-effective solution may found through distributed parallel processing using industry standards such as the Message Passing Interface (MPI) to operate clusters of distributed workstations in parallel. Regardless of the architecture of the parallel system, large problems may be partitioned into a network of smaller problems – each of which may be performed by a single processor. Of course, parallel processing introduces some additional complications. Chief among these are the mechanisms for distributing workload and data throughout the system (particularly so that work is evenly divided). These issues must be explicitly addressed in the development of a parallel system.

There has been some research into applications of such parallel processing methodologies for geostatistical and geospatial analysis. Reviews of work on this topic may be found in Armstrong and Densham (1992), Ding and Densham (1996), Clematis *et al*. (1996), Hambrusch and Khokar (1997), Verts and Thomson (1988), Mower (1992), Deelman and Szymanski (1998), Armstrong (1994, 1995), Armstrong and Marciano (1994) and Li (1992). Although research in this area is embryonic, there has nevertheless been substantial development in recent years.

One of the most computationally intensive aspects of geostatistical analysis involves the search for neighboring points which contribute to an interpolation. In a simple nearest neighbor search, merely the closest point to the desired location must be found. Bilinear interpolation, however, requires *four* nearest neighbors; cubic convolution requires *sixteen*. The search for these additional points increases the complexity of the problem immensely. The impact of such searching is so great that

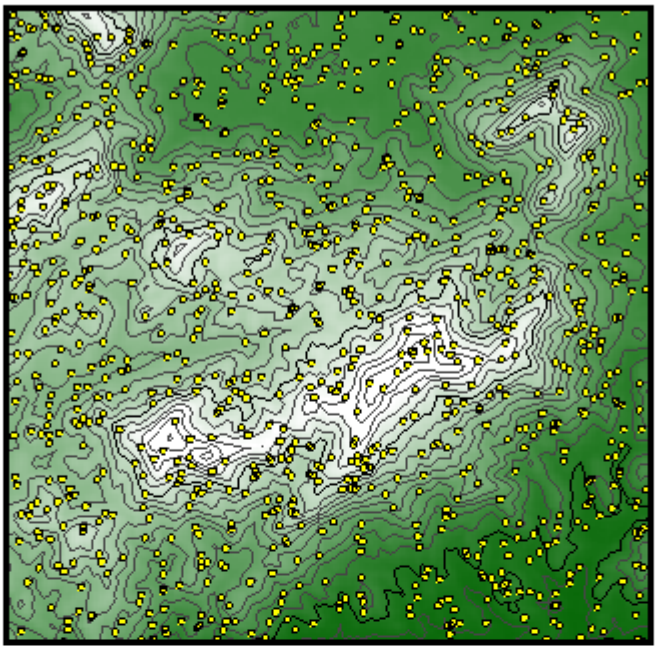
there is a wide body of research which has been developed (Friedman *et al.*, 1975; Lee, 1982; Hodgson, 1989; Clarke, 1990) to most efficiently discover these nearest neighbors. It thus stands to reason that parallelizing the nearest neighbor search may dramatically reduce the computation time required for a geostatistical analysis. As with all parallel systems, there is a signficant overhead incurred due to the parallelization of the system; hence, parallelization may benefit large-scale geostatistical problems but is likely of minimal benefit (and is perhaps even a detriment) to geostatistical analysis of smaller-scale problems.

The primary purpose of this work is to develop an approach which parallelizes the nearest neighbor problem in the plane into one which is addressable by a linear array of distributed processors. The secondary purpose of this work is to construct a distributed parallel system using MPI to implement this approach (applied to inverse distance weighting interpolation) in order to assess its potential benefit (Figure 1.1). Although this algorithm is simplistic and is frequently superseded by more sophisticated kriging or polynomial spline techniques, the methodology developed is fundamental enough to be extensible. An extension of this work to a kriging-based analysis will be discussed at the conclusion as will a more general application to the problem of image projection.

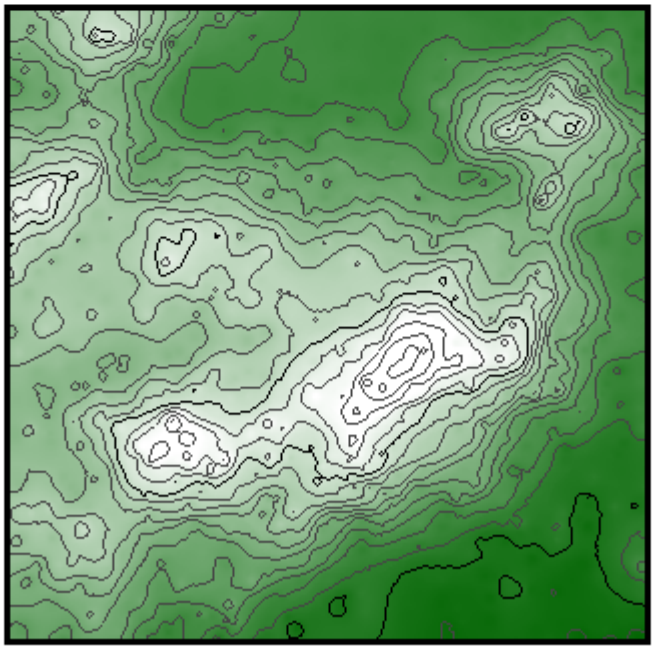
Performance of Sequential Geostatistical Analysis

In order to appreciate the potential impact of parallel techniques for geostatistical analysis, consideration should first be given to sequential implementations of such methods. Several mainstream geographic information systems, spatial analysis and cartographic software packages contain functionality for geostatistical analysis. Environmental Systems Research Institute's Arc/Info software has a sequential implementation of the inverse distance weighting routine used in this work as well as several other related routines included spline surface analysis and

Nearest Neighbor Applications



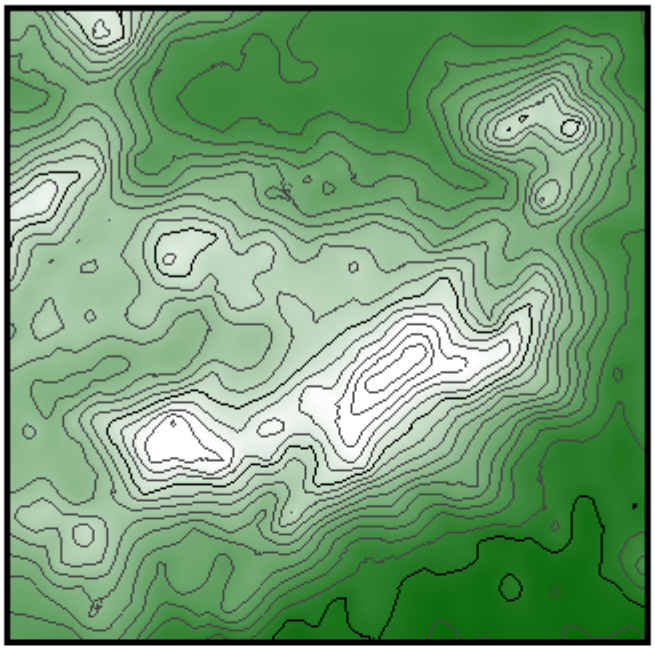
Original Surface and Locations for 1,000 Sampled Points.



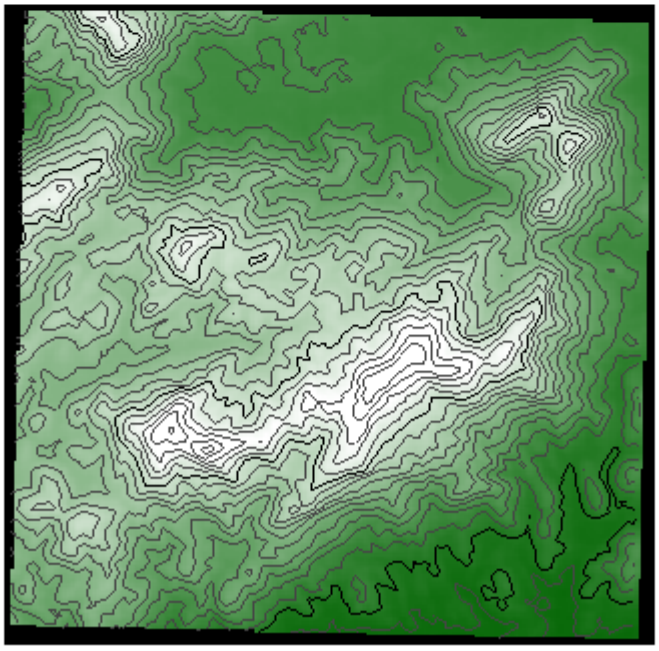
Inverse Distance Weighting Interpolated Surface (with Euclidean distance).

Figure 1.1. Nearest Neighbor Applications.

Nearest Neighbor Applications (Continued)



Kriging Interpolated Surface.



Projected Surface.

Figure 1.1 (Continued). Nearest Neighbor Applications.

kriging. In fact, output from the parallel implementation of the inverse distance weighting routine was compared to output from Arc/Info in order to validate results. Other packages which contain routines for inverse distance weighting and other related geostatistical methods include Environmental Systems Research Institute ArcView and Golden Software Surfer.

The relationship between number of input points, number of output grid cells and surface generation time have been considered for the Arc/Info inverse distance weighting routine (Euclidean distance with 16 nearest neighbors). The interpolation was performed and timed on one 400 MHz processor of a four-processor Sun Enterprise 450 server (Arc/Info is not multiprocessor aware) for numbers of input points from 100,000 to 1,000,000 in multiples of 100,000 and with numbers of output grid cells from approximately 140,000 to 1,250,000 (corresponding to cellsizes decreasing from 360 m to 120 m).

The results show clearly the increasing impact upon surface generation time as the number of input points and the number of output grid cells increase (Figure 1.2). In particular, as both the number of input points and the number of output grid cells increase, surface generation time increases geometrically. For large numbers of input points or output cells, application of geostatistical methods is not practical with sequential techniques. Parallel methods, however, may be applied to address this problem in a practical way.

Parallel Nearest Neighbor Search

Since the nearest neighbor search is one of the most computationally intensive components of a geostatistical analysis (depending upon the particular analysis methodology used), it is therefore a good candidate for parallelization. A brute force nearest neighbor search would require each processor in the parallel cluster to have access to the complete data set. This invalidates the benefit tjat a parallel solution

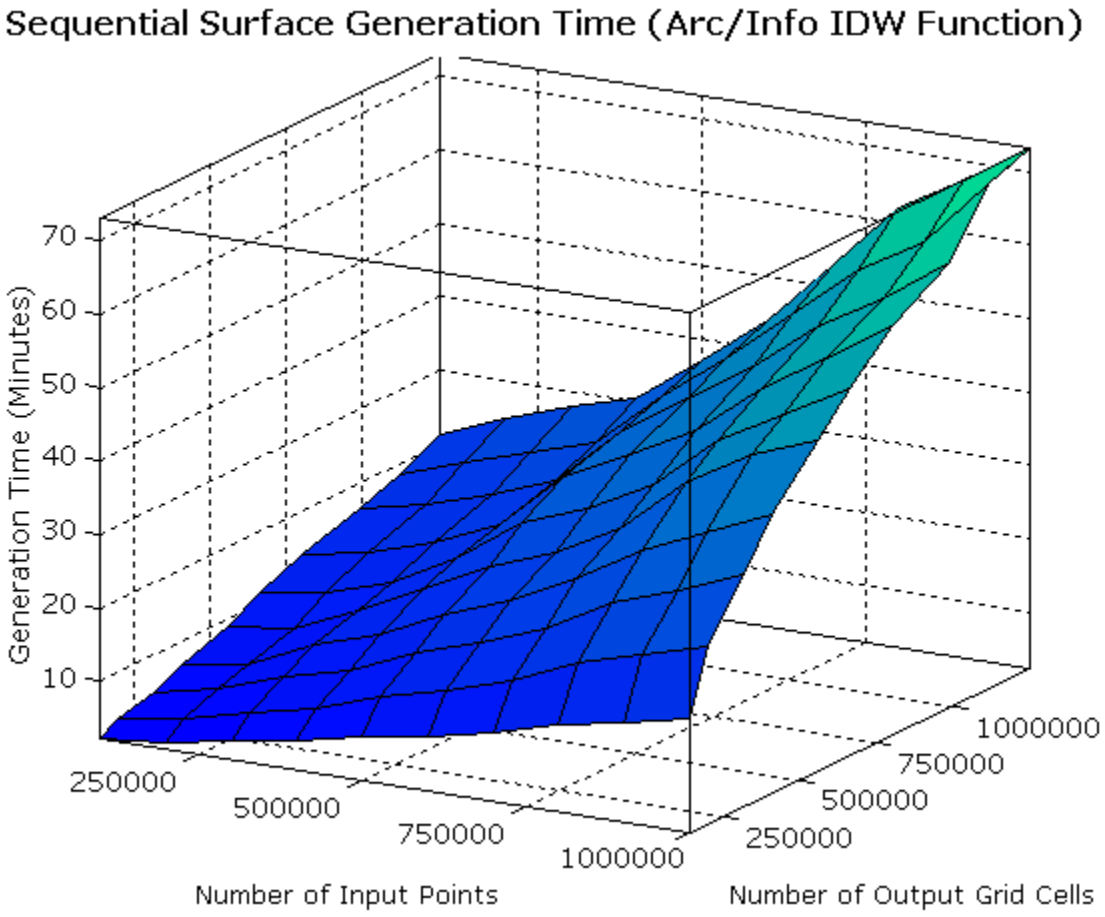


Figure 1.2. Sequential Processing Times for Arc/Info Inverse Distance Weighting (Euclidean distance) with Varying Numbers of Input Points and Output Grid Cells.

might have to perform computation on a dataset larger than that which may be held in memory. A more intelligent method is needed in which each processor requires only a subset of data in memory. Such a method can be found in an algorithm for finding an arbitrary number of nearest neighbors in an arbitrarily dimensional space (Friedman *et al.*, 1975). Adaptation of this method to parallelize the nearest neighbor problem was the focus of this work. Parallelization of the nearest neighbor problem in turn facilitates parallelization of geostatistical methods in general.

The benefit of this methodology has been assessed through implementation of an inverse distance weighting interpolation in software. This software has been developed in C for the Solaris platform, making use of the freely available MPICH implementation of the Message Passing Interface, and contains functionality for measuring elapsed time and number of distance computations. For each of 100 executions of the software on randomly generated datasets ranging from 100,000 to 1,000,000 points with numbers of cells ranging from approximately 140,000 cells to 1,250,000 cells, elapsed time and number of distance computations have been captured for 16 nearest neighbors. Computations were performed on a Sun Enterprise 450 server with four 400 MHz processors running in parallel. These results will show the benefit that such a methodology for parallel nearest neighbor searching may have both to reducing elapsed computation time and reducing the number of required computations.

CHAPTER 2

LITERATURE REVIEW

Searching for Nearest Neighbors

Geostatistics are a class of relatively young techniques which rely on computationally intensive methodologies which have only been readily available since the late 1960s – for example, inverse distance weighting interpolation and kriging. Geoprocessing techniques are also quite computationally intensive and include such methodologies as coordinate system projection.

Geostatistical techniques describe an attribute associated with points distributed throughout a plane in terms of a continuous, parameteric surface (where the x and y parameters correspond to the x and y coordinates of the points in the plane). A frequent application of such geostatistical techniques includes interpolation to produce an elevation surface from sampled elevation values. Other applications are possible, however, and may include the generation of statistical surfaces for any continuous variable such as temperature or precipitation. The well-known kriging technique was derived in order to compute the probability of finding gold ore in a vein based on the location of previous gold ore findings in the mine.

As the name implies, geostatistical techniques make explicit use of the spatial structure of the data that they characterize. Whereas the results of standard statistical techniques report back a descriptive number (such as the mean, standard deviation or t-score) or set of numbers (regression coefficients), the results of geostatistical techniques are spatially explicit surfaces. This surface, which is typically continuous and differentiable, is comprised of a regular lattice of cells – each of which may be

computed in terms of an expression of some subset of input points. Different geostatistical techniques model various assumptions by varying the expression and the determination of the subset of input points. Complexity among geostatistical techniques varies considerably, as do the results of the technique.

Inverse distance weighting interpolation is one of the simplest geostatistical techniques. It is frequently superseded by more advanced techniques such as spline surface interpolation or kriging, but is commonly used as a first approach to interpolating a surface because it may be computed more quickly – comparatively speaking – than a more may complex technique may be. Inverse distance weighting techniques are actually a class of techniques more than a single technique; however, they are similar in that each assumes that neighboring points are more spatially correlated than points which are farther apart. This technique therefore computes the value at a location (in this case, the center of a cell in the interpolation surface) as a weighted function where the contribution of an input point diminishes relative to its distance from the desired location. Points which are farther away are virtually ignored in the computation. The amount of contribution that a point has is defined by the function of distance which defines the technique; one of the most commonly used is Euclidean distance. However, other functions may be used as well including Taxicab distance, log (Euclidean distance) or virtually any other mathematical expression – however, a determination of the accuracy of the expression to adequately describe the trend under consideration should first be made. Although in theory inverse distance weighting techniques may use linear combinations of all points in the input set, in practice this is computationally expensive. Most frequently, some mechanism is used to define a subset of the input which will be used to perform the computation. One method which is frequently used is to take the n closest points to define the subset – $n = 4$ defines a bilinear interpolation and $n = 16$ defines a bicubic interpolation. Another somewhat less commonly used method is to subset all points based upon

their distance from the point to be interpolated – for example, all points within a 25 mile radius. Both of these methods recognize and exploit the fact that for points far away from the interpolation point, the contribution to the overall sum will be minimal. Although inverse distance weighted techniques are fast compared to other geostatistical methods, they suffer from a number of serious deficiencies which often render them unsuitable. First, due to the nature of the weighting, points to be interpolated which lie near input points tend to be expressed in the resultant surface by a "bullseye" pattern because of the input point contributes disproportionately. A variant on the inverse distance weighting methodology known as Shepard's Radial Basis Function corrects for this bullseye pattern by applying a smoothing factor after the interpolation. A second weakness with this methodology is its inability to cope with discontinuous surfaces (for example, cliffs in an elevation surface). Because of this, inverse distance weighting techniques are frequently superseded by more advanced techniques such as kriging. Nevertheless, inverse distance weighting techniques may produce a sufficient surface in a reasonable time frame provided that the set of input points is not too large – a difficulty which can hamper any of the geostatistical methods.

Kriging techniques move beyond simple interpolation by applying regionalized variable theory to better model the error term in the geostatistical surface. Regionalized variable theory assumes that the pattern of spatial variation in the value to be interpolated is does not vary throughout the statistical surface. In simple kriging, this spatial variation is enumerated by a sample semivariogram function of distance. The covariogram is then computed which fits the sampled semivariogram values to a theoretical distribution such as linear, spherical or exponential. The resultant variogram is a monotonically increasing, continuous function of distance. A linear system of equations is then used to solve for interpolation weights for each neighboring point which are a function of the variogram value for the distance

between the points. Construction of the variogram is an intensive process, however, solution of the linear systems for each neighbor is highly computationally intensive. As with inverse distance weighting, the determination of neighbors for which to compute weights may be made either through *n* nearest neighbors or through a search radius; however limiting the number of neighbors for which to reduce weights also reduces significantly the computational complexity of the problem. An excellent review of inverse distance weighting, kriging and several other interpolation techniques is presented in Lam (1983). Various interpolation techniques may also be found in Lam (1983) and Bailey and Gatrell (1995).

In a geoprocessing problem such as projection or resampling, the goal is instead to generate an alternative surface to an input surface, resulting in a change to the surface itself. Projection applies a mathematical transformation to each of the cells in the input surface; each of these transformed cells becomes a sample point in the alternative surface. Some method of interpolation is then used to obtain values for the cells in the alternative surface (typically nearest neighbor, bilinear interpolation or bicubic interpolation using the 1, 4 or 16 nearest neighbors respectively. Resampling produces an alternative surface to the input surface by computing new cells which are of a different resolution than the cells in the input surface. Cells in the alternative surface may technically be smaller or larger than cells in the input surface. There is, however, little or new introduction of information by resampling to a finer resolution. Resampling typically occurs in order to reduce the filesize of a surface – for example, resampling a surface with a 30 m cellsize to a 60 m cellsize may actually reduce the filesize of the surface by 4 times. Again, values for the new cells are computed through some interpolation method, which in turn requires the discovery of *n* nearest neighbors or all neighbors within a specified search radius.

Even from these limited examples, the importance of searching for nearest neighbors can be seen. It is for this reason that such a wide body of literature dealing

with the nearest neighbor problem has been developed. The most simple, intuitive and inefficient method of searching for nearest neighbors is the brute force search in which distances from a point to all other points are computed and exhaustively searched in order to absolutely determine the first *n* nearest neighbors. Computation time for this method increases linearly as the sample size of the dataset increases. Many approaches to finding nearest neighbors that require fewer distance computations have been developed.

An early foundational work was developed by Friedman *et al*. (1975) which statistically determined the maximum number of points for which distance computations would be required in order to determine the *n* nearest neighbors. Moreover, this method presented an ordered approach to determining *which* points required distance computations. The method operates by first sorting sample points along a single dimension and then searching points which are proximal in the sorted direction until a search criteria has been met. Thus, sample points are partitioned in such a way that only proximal points in the sorted direction must be searched for some number of points in the sorted direction (which may also be predetermined). Although not reported by Friedman *et al*., Hodgson (1989) reports a logarithmically increasing computation time as a function of increasing sample sizes.

Hodgson (1989) presents a "learned search" methodology for computing nearest neighbors. Whereas the method presented by Friedman *et al*. (1975) performs each search in the interpolation process independently of all other searches, Hodgon's approach instead learns from previous nearby searches. A sample set of points is partitioned into "sortedcells", or a kind of metacell which includes multiple actual cells. Neighbors for the first grid cell are found by brute force; neighbors for subsequent cells are found subsequently under the hypothesis that the farthest near neighbor must be less than or equal to the distance from the current grid cell to the farthest near neighbor of the previous cell. Hodgson reports that the function of

computation time as the number of samples increases is "almost linear with a small slope". Whereas the Friedman *et al*. (1975) method has a certain degree of independence, the Hodgson method iteratively requires knowledge of previous computations and is highly interdependent.

Clarke (1990) reports a different method than those presented by either Friedman *et al*. (1975) or Hodgson (1989). Unlike these research efforts, however, the Clarke method does not guarantee to find absolutely the *n* nearest neighbors. This method operates by assigning sample points to grid cells as they are read in, averaging sample point values where two or more sample points fall within the same cells. For cells which are empty at the conclusion of this procedure, a neighborhood search begins in which square neighborhoods surrounding the cell are searched until the required number of points are found – however, Clarke reports that it may be the case that more than the *n* nearest neighbors are found. Although not explicitly stated, reduction to *n* nearest neighbors presumably occurs through brute force. Clarke reports only that this algorithm "has been found to be highly effective and is far faster than the brute force method." While not as interdependent as the Hodgson method, the Clarke method does require access to potentially overlapping data in order to accomplish solution.

It has been shown by Lee (1982) that Voronoi diagrams may also be used to solve the nearest neighbor problem, and is in fact quite fast. The solution for finding *n* nearest neighbors requires the construction of an *n* order Voronoi diagram, which may have a linear computation time with respect to the number of sample points, although Lee reports that finding the *n* nearest neighbors is $O(n^2 N \log(N))$ for N sample points. This method also necessitates a high degree of interdependence between points in order to compute the Voronoi diagram.

There are many methods for computing nearest neighbors; each of which has its own strengths and weaknesses. Choice of the nearest neighbor methodology depends highly upon the particular characteristic of the technique.

Parallel Computing

The earliest computing architecture, the Von Neumann machine, consisted of a single central processing unit and a single pool of memory (Xavier and Iyengar, 1998). This design has pervaded computing since first proposed by Von Neumann. In contrast, the parallel architecture consists of many (or many thousands) of processors, each of which may or may not have its own pool of memory. Although technically feasible, dedicated high-end parallel computers are often fiscally infeasible, with costs running into the millions of dollars. In the last few years, multiprocessor machines have become commercially available as well. Typically, however, such multiprocessor systems are not explicitly parallel as there is little commercially available parallel software. Custom parallel software may be developed, either on multiprocessor systems or on distributed parallel systems, through libraries based upon standards such as the Message Passing Interface. In such a way, parallelism may be achieved in a more economically feasible way.

Flynn (1966) identified a taxonomy of four computer architectures. The first of these, the SISD architecture (single instruction, single data) corresponds to the Von Neumann machine. The second architecture, MISD (multiple instruction, single data) is theoretically possible, but is infrequently used and in fact is a special case of the MIMD (multiple instruction, multiple data) architecture. The MIMD architecture corresponds to multiple independent operations on varying data streams; an application may be found in solutions to games of strategy such as chess (Akl, 1989). Designing MIMD programs can be difficult as both data and work partitioning must occur. Nevertheless, the MIMD architecture is highly powerful.

The fourth architecture, SIMD (single instruction, multiple data) is used in parallel systems where the same operation occurs on subsets of the input data (for example, sorting) with parallel processors under the control of a single control unit.

Communication in parallel systems may take place either via shared memory or an interconnection network (Akl, 1989). In shared memory systems, a processor wishing to communicate with another processor may write data to some place in memory and pass a pointer to that memory to the second processor. In interconnected systems, however, memory is distributed so that passing data to a second processor requires actually passing the complete set of data. Shared memory systems are typically more expensive than interconnection networks and are usually found in massively parallel systems, although multiprocessor systems parallelized via MPI may also conceivably make use of shared memory on a much smaller scale. Costs for shared memory may be so prohibitive in fact that even on massively parallel systems, memory is not completely designed but is instead divided into blocks which are connected through an interconnection network. Likewise, distributed parallel systems are also connected through an interconnection network.

There are a number of network architectures which are commonly used in interconnection design (Akl, 1989). A linear array connects a series of P processors together in such a way that processor p may only communicate with processors $p - 1$ and $p + 1$ through a two way connection. Processors 1 and P only communicate with the single processor above or below it, respectively. In contrast, the two-dimensional array (or mesh) connects memory blocks through a matrix. In this model, each processor may communicate with the four nearest processors above or below it (either row-wise or column-wise). As with the linear array, processors on the boundary may communicate with fewer processors than interior processors – typically three neighboring processors, but the four cells which are in either the first or last column and the first or last row may only communicate with two neighboring processors. The

linear array and mesh networks enjoy an advantage in that the connection "length" between each processor is uniform, which in turns endows scalability where additional processors may be added to the system easily. The remaining interconnection network architectures, however, do not have such a property.

The tree connection architecture is structurally identical to a binary tree whereby each parent processor has two children (starting at a root processor) and each child has a single parent. Each processor may communicate only with its children and parents. The closer a processor is to the root, the longer its interconnection line is between it and its parent. A cube (or hypercube) connection extends the mesh concept to a higher q dimensional space by connecting each processor to its q neighbors. The hypercube architecture is frequently used in parallel systems. In addition to these, there are several other types of interconnection networks, including pyramid networks and star graphs (Xavier and Iyengar, 1998).

A particular problem common on parallel systems involves the input and output of data to and from the system. Four classes of I/O may be readily defined. The least flexible model is the exclusive read, exclusive write whereby only a single processor may access a memory location at a single time. A more flexible and more common approach is the concurrent read, concurrent write model where data may be read from a memory location by all processors, but only a single processor may write to the memory location at a time. Less commonly used is the exclusive read, concurrent write approach. Because access problems are more inherent to writing and not reading, this approach is not typically used. Finally, concurrent read, concurrent write operations allow all processors to access a memory location either through read or write operations at the same time. Concurrent write operations are difficult to manage in that determination must be made in an attempt to simultaneously write disparate values to the same memory location as to which process should succeed.

There are many issues which must be considered when developing a parallel system, including these and others. Design of the parallel system will be based in part upon the design decisions made in these issues as well as upon the nature of the problem to be solved by the parallel system.

Parallel GIS

Although research in parallel GIS is not a new topic, it is only recently that the hardware has been readily available to support it. Research topics in parallel GIS are as varied as subject areas in GIS itself, but can be loosely categorized into architectural issues and applications. Architectural issues range from dedicated hardware and distributed parallel systems to software, programming, high performance computing, database systems, data models and memory allocation and management. Applications include spatial modeling, geoprocessing and analytical tools.

As with sequential GIS systems, parallel GIS systems require suffificent infrastructure in order to efficiently operate. This infrastructure may take the form of a dedicated machine or a distributed parallel network of machines (or perhaps a combination of both, such as with a cluster of multiprocessor servers). An understanding of the benefits and drawbacks associated with each model is key to making good decisions in the implementation. A summary of these issues is presented in Sawyer (1998a) along with details about parallel computer systems not usually treated in GIS literature. Specific emphasis is placed upon the impact of these issues to GIS computing. Clematis *et al*. (1996) address the issue of distributed parallel computing in greater detail, dealing specifically with libraries for interconnection and with applications utilizing such networks. A general treatment of parallel GIS architectures can also be found in Verts and Thomson (1988).

No less important than hardware is the issue of software. There is little commercially available parallel software and less parallel GIS software; what software is available is typically designed for massively parallel systems. For virtually all

parallel GIS applications, custom software must be designed – typically with tools such as PVM or MPI (Sawyer, 1998b; Trewin, 1998) although efforts are underway to develop more readily available libraries of parallel GIS programs (Mineter *et al.*, 1998a). Development of parallel algorithms, however, differs drastically from development of sequential algorithms – even for aspatial problems. Research into the design of parallel spatial algorithms is particularly important to the development of parallel GIS (Dowers *et al.*, 1998a and b; Armstrong and Densham, 1992; Ding and Densham, 1996).

The principal factor driving the development of parallel GIS tools is to optimize processing times, particularly for computationally complex problems. As data collection procedures have improved, data volumes have grown exponentially. Tools that once may have been computationally tractable on much smaller data volumes are now becoming intractable and new approaches to the solution of spatial problems are necessary. Moreover, there are many spatial techniques for which computational complexity is very high – even for small data volumes. The impact on these techniques by large data volumes is overwhelming. The field of computing that attempts to address such computationally intensive problems is referred to as high performance computing and efforts are underway to assess the importance of this paradigm to GIS (Armstrong, 1994, 1995). Related to this issue of data volumes is the problem of storing this massive amount of data. This issue is being dealt with on two fronts. Parallel implementations of large scale database management systems for GIS are currently being researched to store for ready access these large volumes of data (Wilkinson, 1998). On a much smaller scale (but no less important), efficient data structures for parallel GIS are also under investigation (Dowers, 1998; Mineter, 1998a and b; Mineter *et al.*, 1998b) as are mechanisms for interoperability between such data structures (Sloan, 1998; Mineter, 1998c). Issues of efficient memory allocation are also under investigation both for shared memory architectures (Shekar *et al.*, 1996) and distributed memory architectures (Hambrusch and Khokar, 1997).

Of particular importance is the performance of the system (Dowers *et al.*, 1991), and various load balancing strategies for spatial systems have been developed to maximize this performance (Deelman and Szymanski, 1998; Shekar *et al.*, 1998).

Research into the applications of parallel spatial systems is no less varied. As would be expected with GIS, research is divided among raster and vector approaches. Each of these approaches in turn is characterized by research that is either theoretical in nature or that is more problem domain specific.

Theoretical vector applications of parallel GIS may be found in Roche and Gittings (1996), Mower (1996), Li (1992), Harding *et al*. (1998) and others. Of particular interest to this research, raster investigations may be found in Armstrong and Marciano (1994), Densham and Armstrong (1998) and others. Problem specific approaches and spatial modeling may be found as well in Xiong and Marble (1996), Magillo and Puppo (1998) and Wilkinson (1998).

Parallel GIS is a young but rapidly expanding field with the potential to revolutionize the GIS industry. Indeed, Dangermond and Morehouse (1987) indicate that parallel processing is perhaps one of the most important hardware trends for GIS. Parallel computing for GIS has implications that reach far beyond the applications now achievable with current technology. Research thus far is exciting, but is just the tip of the iceberg.

CHAPTER 3

DEVELOPMENT OF APPROACH

Review of An Algorithm for Finding Nearest Neighbors

One of the most computationally intensive components of geostatistical analysis involves the search for neighboring points to contribute to the interpolation. A brute force search for k nearest neighbors involves computing and comparing the distance for each and every prototype point in the input dataset (Figure 3.1). If the surface to be interpolated contains g grid cells, this requires $k * g$ distance computations. For very large prototype datasets and very finely resolved surfaces, this number may be very high (for example, interpolation of a surface with 1,000,000 grid cells from 1,000,000 prototypes results in 10^{12} required distance calculations). Moreover, a parallel solution to the brute force nearest neighbor search requires that each processor have full and complete access to the entire set of prototypes. For shared memory parallel systems this may not be an issue, however for distributed memory parallel systems this can be a serious deficiency. The full set of prototypes may be made available to each processor in the parallel cluster through one of two approaches. First, the full set may be loaded into memory on each of the processors. This approach negates the ability of the parallel system to perform geostatistical analysis on very large prototype sets exceeding the memory capacity of the system (a very desirable characteristic). The second technique involves loading parts of the set of prototypes into memory on each of the parallel processors and sharing prototypes through interprocess communications. This method results in a highly expensive set

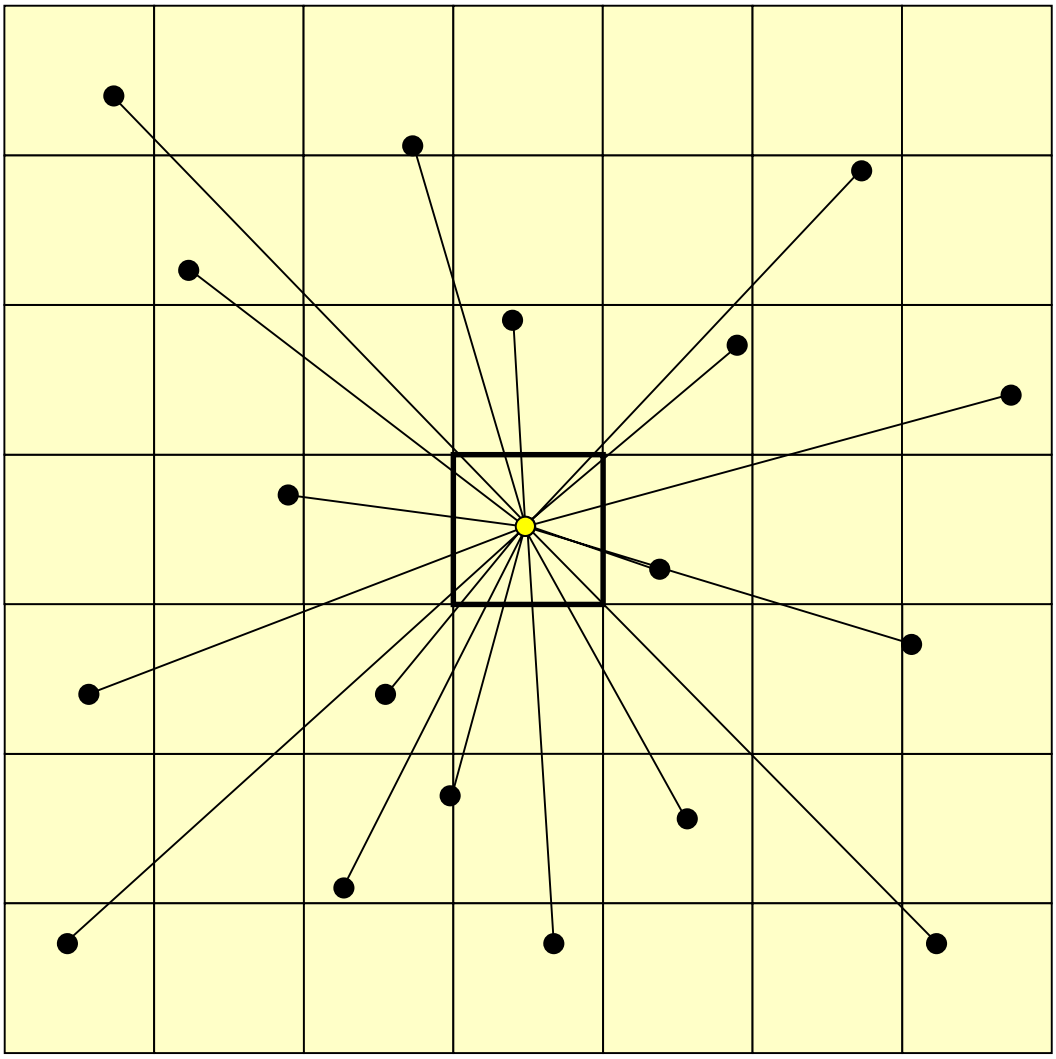


Figure 3.1. Brute Force Nearest Neighbor Search.

of communications back and forth between processors incurring drastic overhead to the parallel system.

Figure 3.2 illustrates the impact that a parallel implementation of a brute force nearest neighbor search would have upon a parallel system. Interpolation of each cell requires the system to access not only data which are stored locally (prototypes within the yellow cells) but also data which are stored remotely (prototypes within the blue and green cells). Each request to a remote processor incurs overhead on the parallel system and requests must be made repeatedly for the same data for each cell in the interpolated surface.

There have been many algorithms developed to more efficiently discover nearest neighbors. Some of these techniques rely on partitioning the problem (a desirable approach for consideration of a parallel solution). Other techniques rely upon learned searching whereby the results of previous searches are retained and used to refine further searches. Although this is a sound approach, it is difficult to implement in a distributed memory parallel system as these results must be shared back and forth among processors.

A classic method for finding nearest neighbors was proposed by Friedman *et al*. (1975) and has been the basis for much subsequent work. This work proposes a flexible approach for computing k nearest neighbors in a d-dimensional space for N prototypes with an expected upper bound – for the purposes of geostatistical analysis, a value of 2 for d suffices. The algorithm operates by first sorting all of the prototypes along and projecting prototypes to one of the axes, reducing the problem from a d-dimensional problem to a 1-dimensional problem. For each cell in the surface to be interpolated, a seed prototype is found which is closest in projected distance to the center of the cell and the d-dimensional distance is computed. This d-dimensional distance is then used as the search radius to search subsequent points. Any points which are farther away in projected distance than this d-dimensional distance must

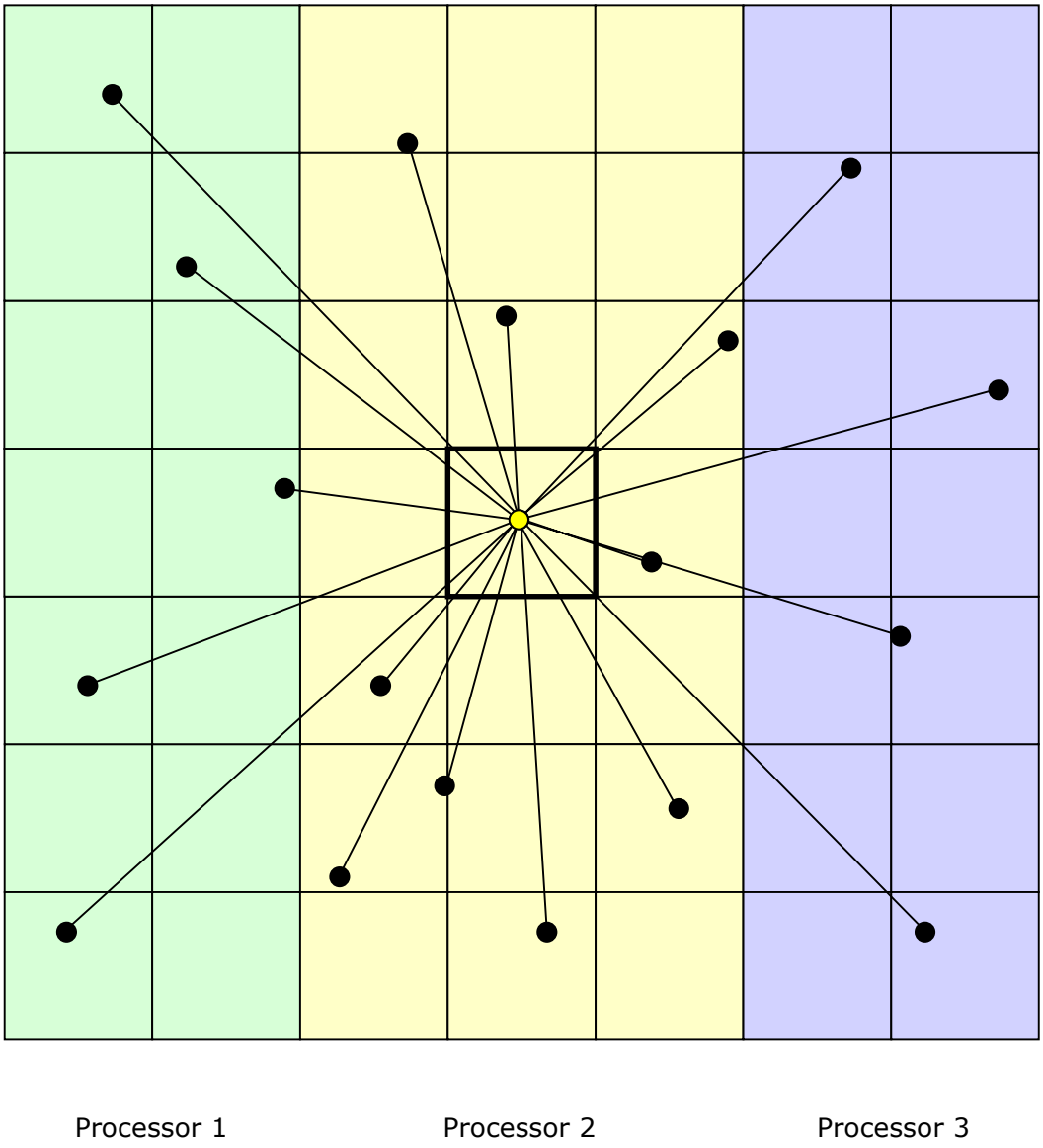


Figure 3.2. Parallel Brute Force Nearest Neighbor Search.

necessarily be farther away in their own *d*-dimensional distance and are therefore not candidates to be nearer neighbors. The search continues until the prototype currently being searched is farther away in projected distance than the d-dimensional search distance (Figure 3.3). To search for *k* nearest neighbors, this process continues iteratively, searching for neighbors within the search radius, recomputing their own *d*-dimensional distance and using this instead as the modified search radius.

There are several advantages to this approach. The most obvious advantage is that only a subset of distances to prototypes must be computed. Moreover, prototypes are searched in increasing projected distance from the center of the cell to be interpolated, thus bestowing a natural ordering to the structure of the prototypes which is not present in the brute force nearest neighbor search (it is known that only prototypes close in projected distance must be searched). Finally, and perhaps most importantly, Friedman *et al*. determined that in fact the upper limit of prototypes that must be searched is known and in fact can be expressed as

$$E[n_d] \le \pi^{-1/2}[kd\Gamma(d/2)]^{1/d}(2N)^{1-(1/d)} \tag{1}$$

which can be reduced for geostatistical analysis in the plane(with $d = 2$) to

$$E[n_2] \le \frac{(2k)^{1/2}}{\pi^{1/2}}(2N)^{1/2} \tag{2}$$

since

$$\Gamma(2/2) = \Gamma(1) = 1. \tag{3}$$

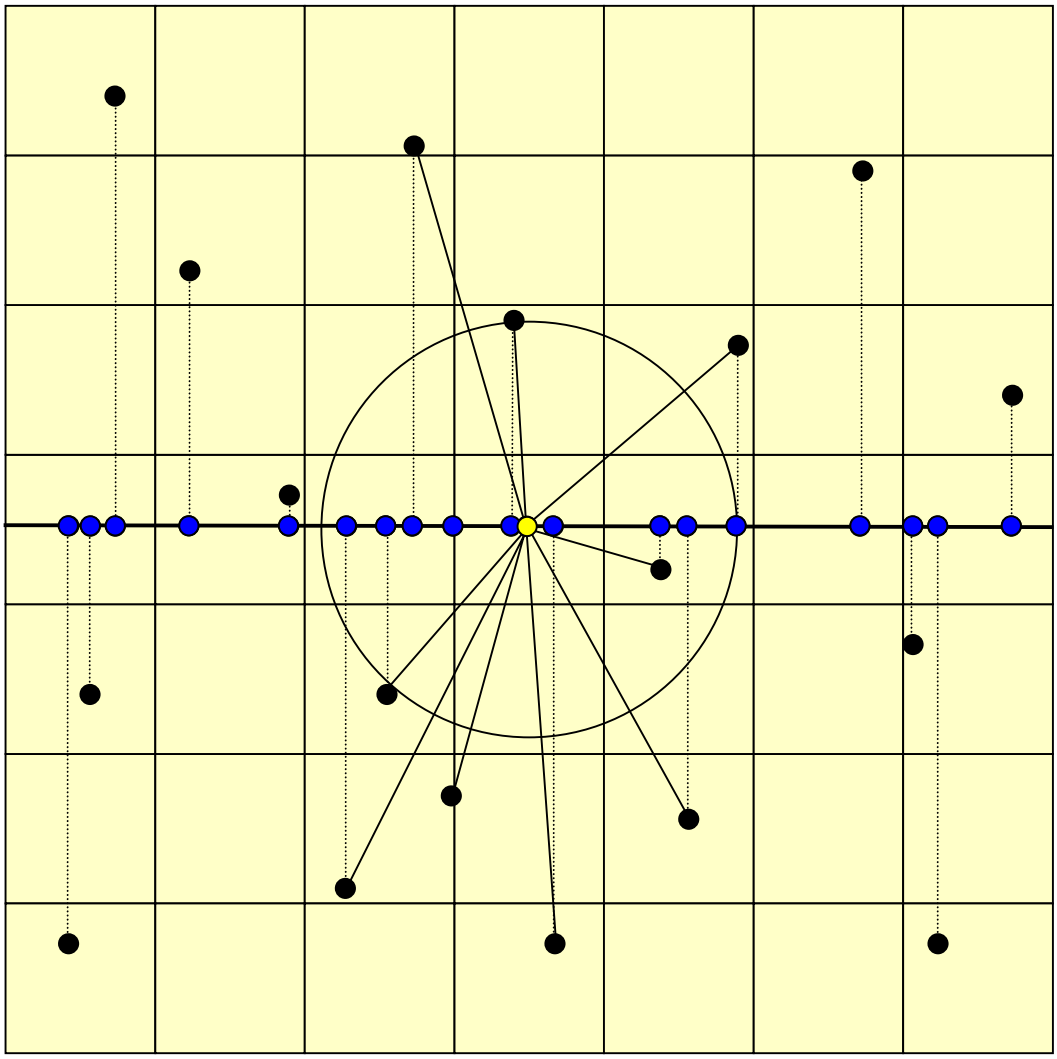


Figure 3.3. Intelligent Nearest Neighbor Search (Friedman *et al.*, 1975).

Thus the exact subset of prototypes which must be searched in order to find the k prototypes is known. It is also the case that the number of prototypes which must be searched increases logarithmically (and not linearly) with an increasing total number of prototypes, making the algorithm more suitable for large prototype sets than for small prototype sets as the ratio of total input prototypes to prototypes which must be searched decreases exponentially. Figure 3.4 illustrates the total number and proportion of prototypes which must be searched for $k = 16$ and $d = 2$. The only significant drawback to this algorithm is the additional overhead of sorting the prototypes in one of the dimensions. For low numbers of prototypes, this overhead may overshadow any benefit gained from the method (for $N = 100$, $k = 16$ and $d = 2$ the ratio of prototypes to be searched is 45% so the overhead of presorting the points would probably negate any improvement gained from the application of the algorithm. However, for $N = 1{,}000{,}000$, $k = 16$ and $d = 2$ the ratio of prototypes to be searched is only 0.45% and thus the overhead of presorting is probably minimal as compared to the benefit gleaned from the algorithm).

A Parallel Approach to Searching for Nearest Neighbors

The method by Friedman *et al.* (1975) may be adapted to parallelize the nearest neighbor search in the plane into a problem which is can be implemented on a linear array of P parallel processors with distributed memory. This method is flexible enough that the choice of P is nearly arbitrary with a few caveats which will be discussed later.

A linear array of P parallel processors is a parallel architecture such that each processor p may communicate only with processor $p-1$ and processor $p+1$. Processors 1 and P each only may communicate with a single processor (processor 1 communicates with processor 2 above it and processor P communicates with processor

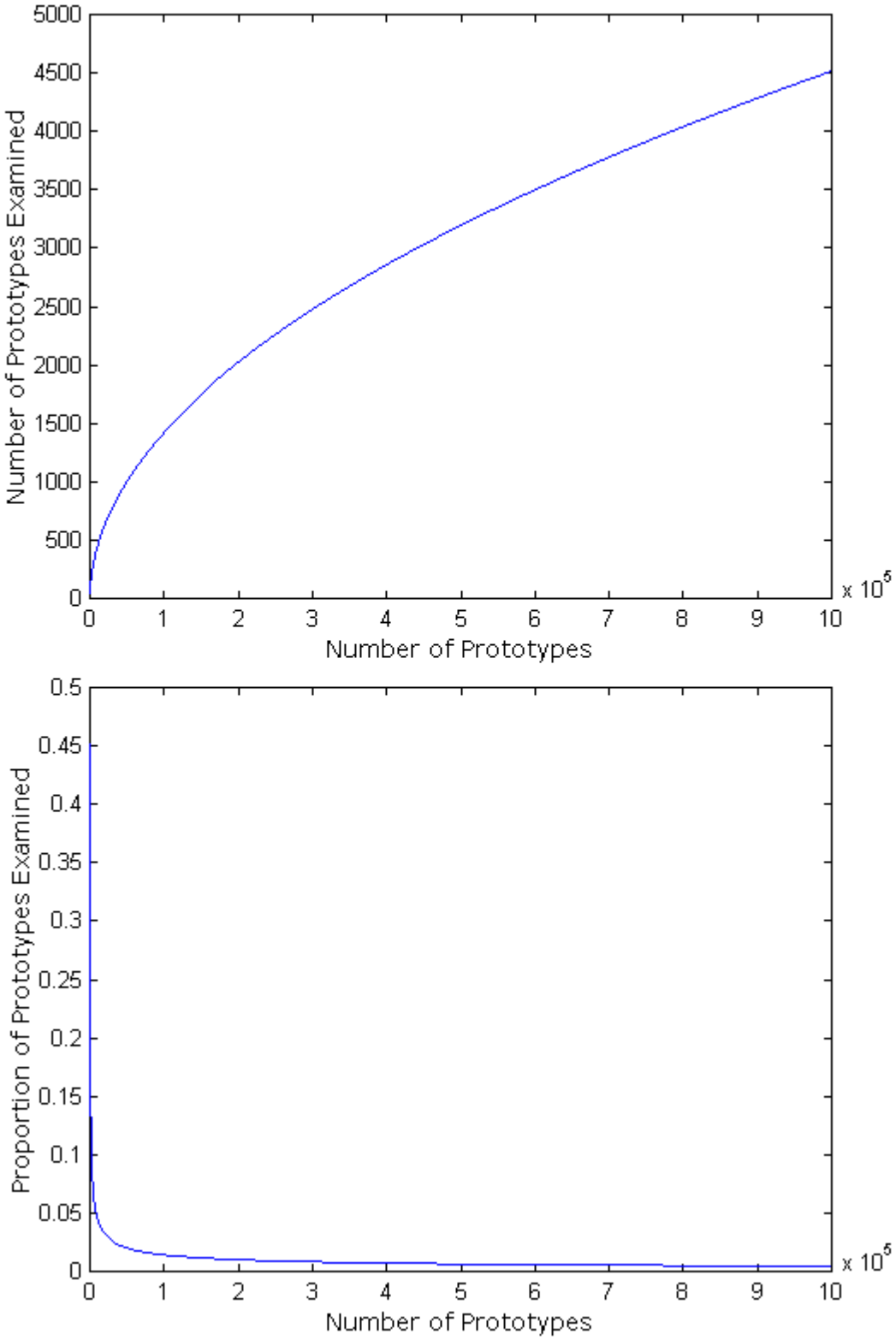


Figure 3.4. Expected Number and Ratio of Prototypes to Search (Friedman *et al.*, 1975).

that it is guaranteed that a process p will only at most need to process prototypes from processes $p - 1$ and $p + 1$ (Figure 3.5), the primary objective being the minimization of interprocess communications. Using the Friedman *et al*. (1975) method, there are two principal components to performing the nearest neighbor search: (1) presorting the data and (2) finding nearest neighbors.

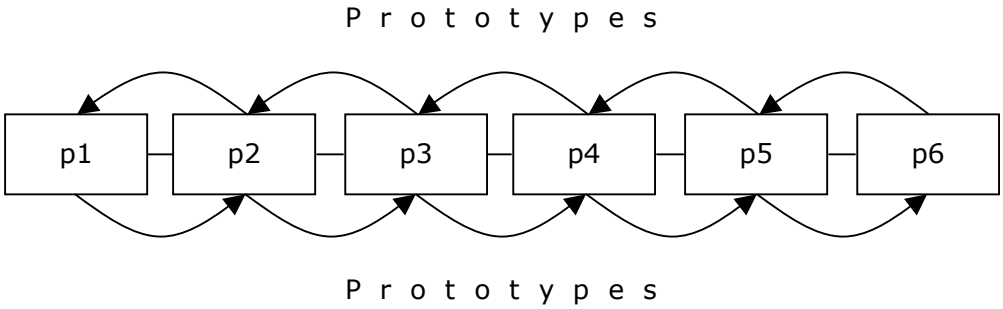


Figure 3.5. Linear Array of Parallel Processors for Nearest Neighbor Searching.

Given a parallel algorithm for presorting the points which is implementable on a linear array, such as a parallel merge-split sort, it turns out that the properties of the Friedman *et al*. algorithm insure that interprocess communications can be constrained to processors $p \pm 1$ from processor p. In point of fact, it can actually be insured that interprocess communications will be limited exclusively to the parallel sort! The reasons for this are detailed as follows.

It should be noted once more that the algorithm by Friedman *et al*. operates on a set of prototypes which are sorted in a single dimension. This fact is important because it means that distance comparisons and sorting must occur only upon a single coordinate.

Consider an array of projected coordinates of prototypes (Figure 3.6).

5	8	2	11	1	7	10	12	6	9	4	3

Figure 3.6. Sample Array of Projected Prototypes.

Parallel merge-split sorting operates by partitioning the prototypes into subsets among processors, sorting each subset locally, and then merging results by communicating and merging iteratively with processors above and below it (Figures 3.7 – 3.11 illustrate this process with four processors). Although the specific algorithm used to sort the data is not important, the end result is that each processor contains a sorted subset of the data prototypes are also sorted across processors.

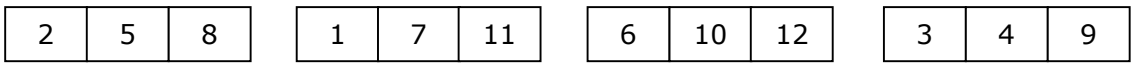


Figure 3.7. Intermediate Local Sort in Parallel Merge-Split Sort.

Figure 3.8. Parallel Merge Step 1.

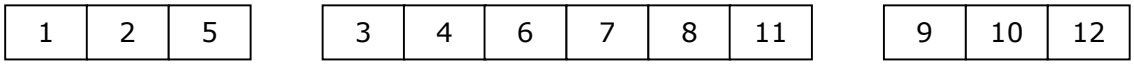


Figure 3.9. Parallel Merge Step 2.

Figure 3.10. Parallel Merge Step 3.

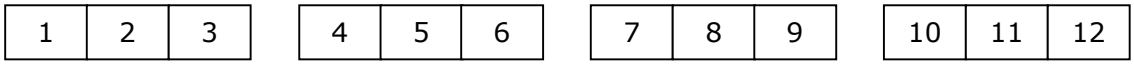


Figure 3.11. Parallel Merge Step 4.

Now consider the projected coordinates of the center of five cells in a surface to be interpolated (Figure 3.12) and suppose that it has been determined by (1) that

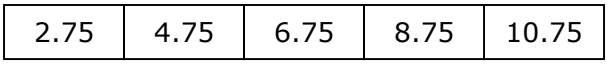


Figure 3.12. Sample Array of Projected Cell Centers.

the maximum number of prototypes which must be searched is 2 (the selection of this number is purely arbitrary and is selected only for example). In order to proceed with the parallel nearest neighbor search, the prototype seeds for each of the cell centers must be found (the prototype seed is the prototype which is closest in projected distance to the cell center). The partition of work then is (Figure 3.13)

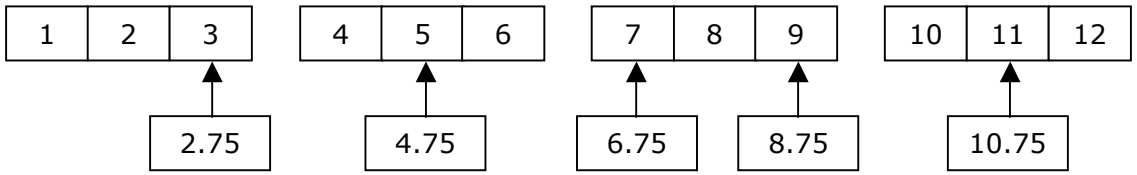


Figure 3.13. Location of Prototype Seeds to Partition Work.

so that each processor has its particular assignment of work (processor 1 will interpolate cell 1, processor 2 will interpolate cell 2, processor 3 will interpolate cells 3 and 4 and processor 4 will interpolate cell 4). Processing may begin on each processor independent of work done on all other processors.

Now consider the interpolation task on processor 1 (Figure 3.14). As stated

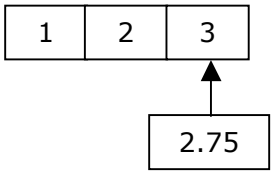


Figure 3.14. Nearest Neighbor Search on Processor 1.

earlier, the maximum number of prototypes which must be searched for this example is 2 – although it is not known whether it will be 2 prototypes above the seed, 2 prototypes below the seed or 1 prototype above and 1 prototype below the seed. Considering the interpolation task on processor 1, it can be seen that processor 1 contains data for 2 prototypes below the seed, but no prototype above the seed. This is a situation which would necessitate interprocess communications without the a

priori determination by equation (1). Because it is known before nearest neighbor searching begins that 2 prototypes above and below the seeds will be required, the merge operation may complete instead by also sending enough prototypes above and below the set to satisfy (1) so that all prototypes may be accessed locally (Figure 3.15).

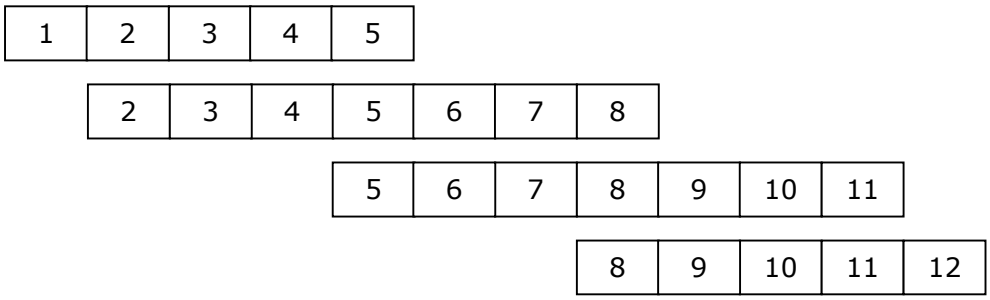


Figure 3.15. Intelligent Prototype Partitioning Based Upon Equation (1).

Returning to the task on processor 1 (Figure 3.16), all of the prototypes are stored locally to discover the nearest neighbor to the cell center of interest. Hence, no additional interprocessor communications are required to find the nearest neighbor to the cell center.

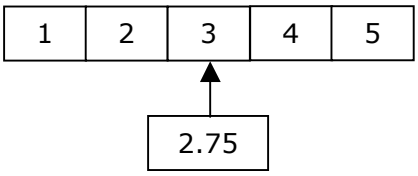


Figure 3.16. Intelligent Nearest Neighbor Search on Processor 1.

Given this simplistic example, it may seem that each processor must store so many extra prototypes that the data division is invalidated. While this may be the case for relatively small prototype set sizes (with 16 nearest neighbors, fewer than 500 points will require more than a 20% overlap), recall Figure 3.4 which showed a logarithmically increasing number of prototype points and an exponentially decreasing

proportion of prototype points which must be searched in order to absolutely determine the nearest neighbor. Consider, for example, a prototype set of 100,000 points on a linear array of 10 processors (so that each processor must hold 10,000 prototypes each) searching for 16 nearest neighbors. By (2), the number of "overlap" prototypes is 1,010 – so that each processor must hold 12,020 prototypes to insure that no interprocess communications are required (a proportion of 12% of the prototypes per processor). For 1,000,000 prototypes under the same condition, the number of "overlap" prototypes is 3,192 – for a total of 103,192 prototypes per processor (a proportion of 10.3% per processor). Clearly as the number of prototypes increases, the efficiency of the technique also increases.

Figure 3.17 shows the implementation of this technique for the sample surface and prototypes which were presented in Figure 3.3. Note that computation is constrained to the processor, as outlined in this methodology.

An additional property of this method which is well worth mentioning is that for surfaces which are regular with respect to the prototypes (as is the case with geostatistical methods but which is not the case with projection techniques), the seeds are exactly the same as they vary across cells in the direction which is not the projected direction. More concretely, if the prototypes are projected along the x axis, then seed points will be calculated for each cell in the row (or raster). Prototypes will be in exactly the same x position from raster to raster. Hence, seeds need only be computed once and can then subsequently be reused.

An efficient methodology which capitalizes on the raster structure of the grid surface is to sort the prototypes in the x direction and to assign a subset of the sorted points across the processors. Once prototype seeds are found for each of the cells in the first raster, neighbors can be computed on the processor for each seed which falls within the data domain of that processor (that is, neighbors can be computed column by column for columns in the problem domain of a particular processor). Once all of

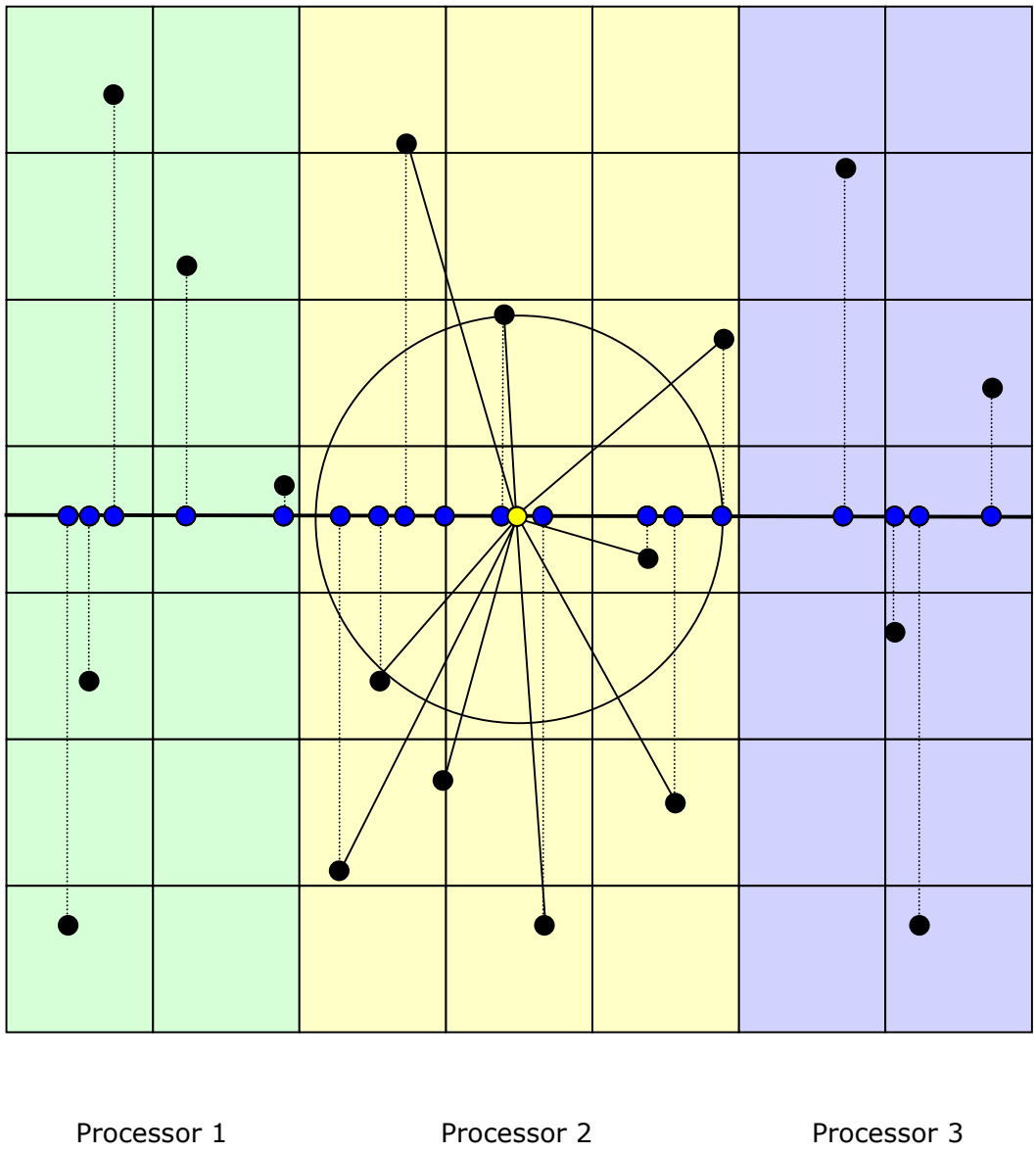


Figure 3.17. Parallel Intelligent Nearest Neighbor Search.

the columns in the raster have been computed, processing may begin on the next raster (Figure 3.18). This process continues until the entire surface has been computed. The advantage of sorting on the x direction, rather than the y direction, and then processing first by rasters and then by columns is that complete partial rasters may be completed in sequence. This may benefit disk operations in that a complete partial raster may be written back the grid file rather than one at a time.

A final caveat in the selection of *P*. Although this method is insensitive to *P* and although more processors will bestow more computing power, there is an upper bound to the number of processors which will be beneficial to the system. In particular, if the ratio of prototypes to processors results in an "overlap" set which is larger than the legitimate number of processors on the system, this approach will not behave predictably. For example, consider a search for 16 nearest neighbors on 10,000 prototypes with 1,000 processors. Equation (1) predicts a 452 prototype searches, however there are only 100 prototypes per processor. (Arguably it could be said that since there are 100 prototypes per processor, only 46 prototype searches are needed per processor, however, even at this it requires searching most of the prototypes in the set and greatly reduces efficiency). The selection of *P* should be done carefully to avoid a situation such as this.

Figure 3.19 depicts the framework for parallel nearest neighbor search which minimizes interprocess communications. Note in particular that the only interprocess communication required is during the sort and "padding" phases of the operation. Outside of this communication, each processor may operate independently.

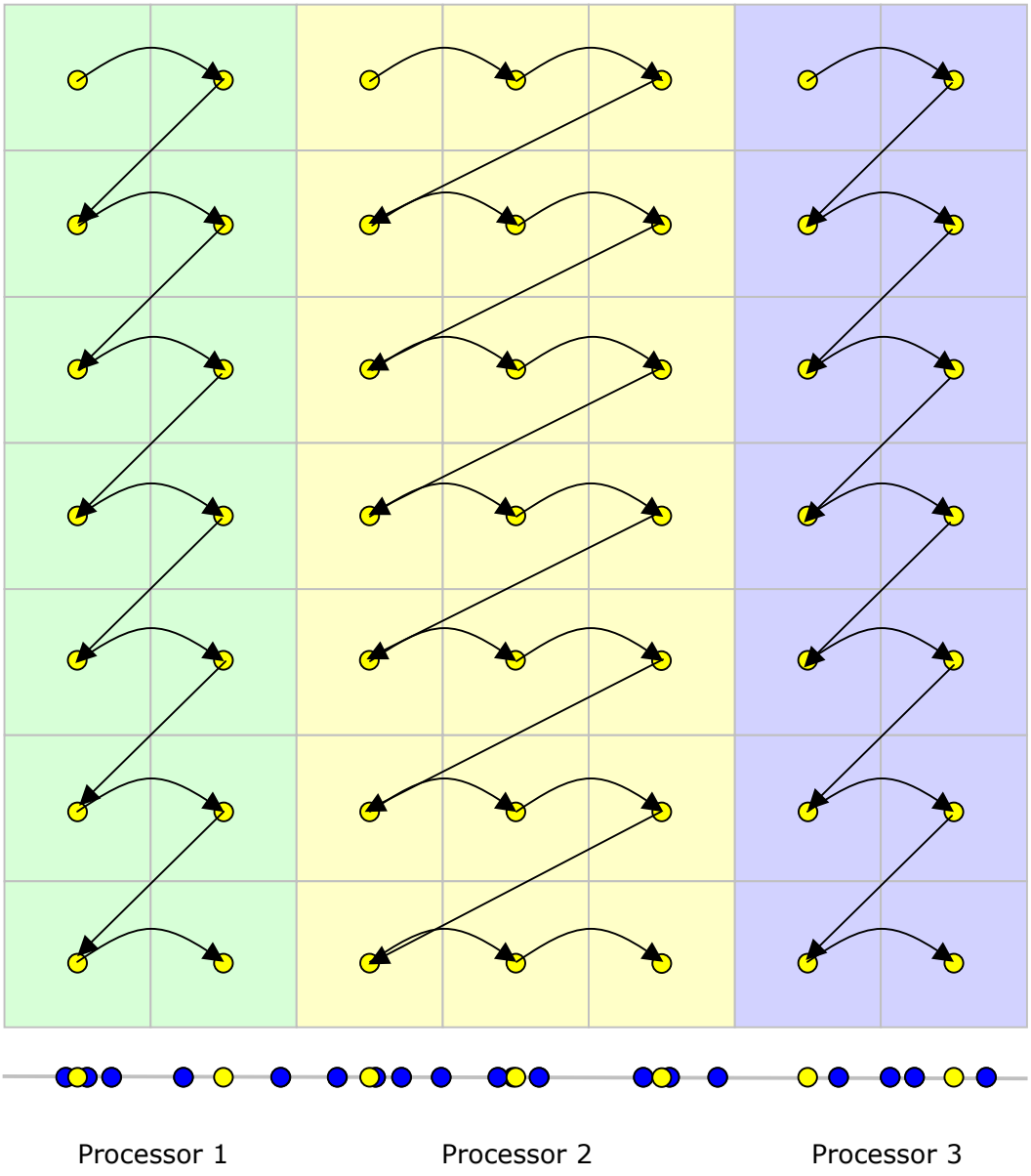


Figure 3.18. Order of Operations for Parallel Intelligent Nearest Neighbor Search.

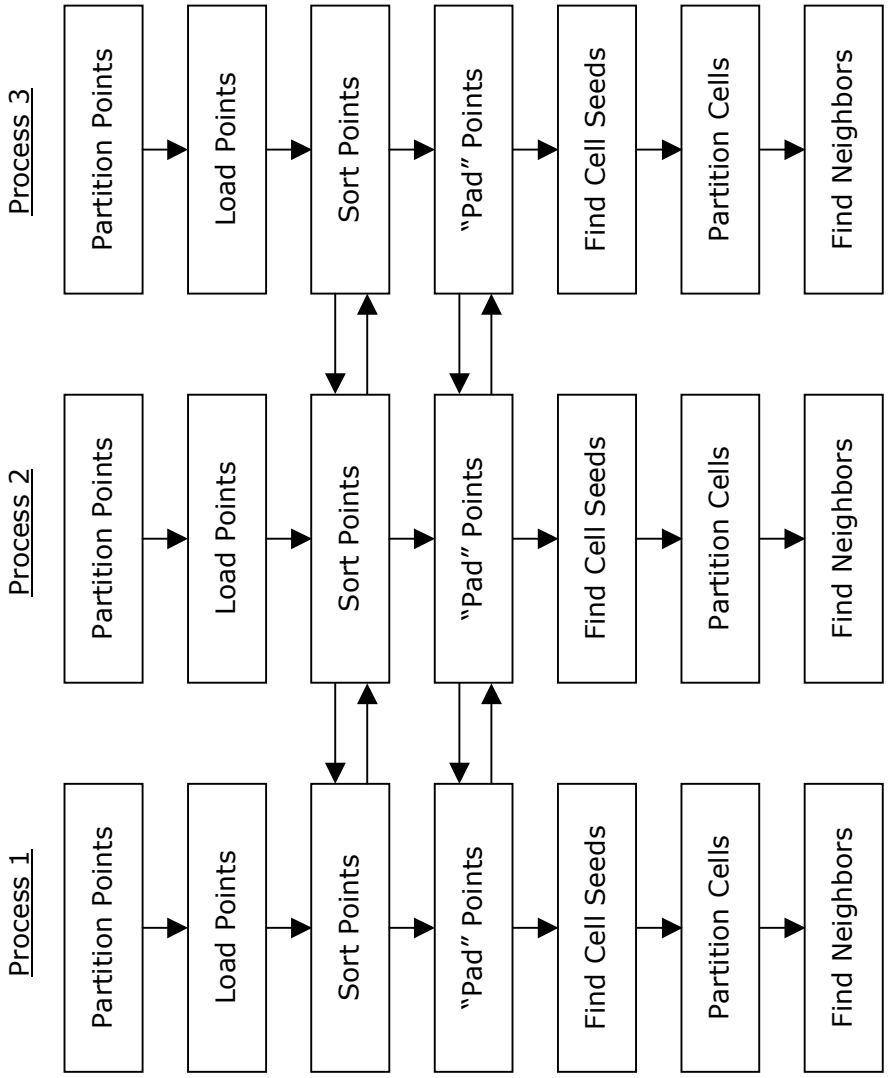


Figure 3.19. Framework for Parallel Nearest Neighbor Search.

CHAPTER 4

APPLICATION TO IDW INTERPOLATION

In order to demonstrate the effectiveness of this approach for searching for nearest neighbors in parallel, an application to inverse distance weighted interpolation was implemented using MPI to provide parallel communications capabilities on a four processor Sun Enterprise 450 server. Point data used in the development and execution of this application were extracted from a large Digital Elevation Model, also generated in the course of this work.

Study Area

In order to provide sufficient data to demonstrate the effectiveness of this approach, a large Digital Elevation Model was developed from which elevation data (points) could be sampled. To create the Digital Elevation Model, individual 7.5 minute (30 meter) DEM images were downloaded from the Georgia State Data Clearinghouse and from the United States Geological Survey and reprojected to Lambert Conformal Conic projection. Distance units were converted to meters and elevation units to inches. The DEM images were then mosaicked together and clipped to a rectangular extent which approximately bounds the Chattahoochee National Forest in order to constrain sample data locations. The resultant DEM image was then saved to 16-bit GeoTIFF format. The Digital Elevation Model encompasses a rectangular area surrounding the Chattahoochee National Forest and covers 11,194 square miles (11.7% of the area of the state of Georgia). This area of the state of Georgia is in the foothills of the Appalachian Mountains. Elevation values

captured in the Digital Elevation Model range from 522 feet above sea level to 5,430 feet above sea level with a mean elevation of 1,427 feet (standard deviation of 694 feet). In some sections of the elevation model topography is relatively constant while in others it varies widely (for example in and around Talullah Gorge in the northeastern region of the study area). Figure 4.1 shows the study area and the Digital Elevation Model for this study.

Generation of sample data was completed programmatically through the *createpoints.c* program developed in C (gcc compiler version 2.95.2 on the Solaris 2.6 platform); Appendix C contains a listing of this program. Using this program, a specified number of random points may be bilinearly interpolated from a Digital Elevation Model in GeoTIFF form and save in a binary format which is readable by the parallel software subsequently developed (the format of this binary file is defined as a Points structure in the *geodata.h* header file and the associated *geodata.c* listed in Appendices A and B). Random distributions are generated independently for the x and y positions of the points using the random number generator rand() seeded with current time in seconds. The *convert.c* program (Appendix D) contains functionality for reading the binary points file and converting it to Shapefile format (with the shapelib library, version 1.1.2); this program may also convert the points to a comma-delimited text file and may convert the binary grid file which is the output of the parallel software to GeoTIFF format (using the libgeotiff version 1.2.4 library). Conversion routines to and from standard spatial data formats as well as definition of the Grid structure are also stored in the *geodata* module and header files. Both the shapelib and the libgeotiff libraries are maintained by the Open GIS Consortium.

Parallel IDW Software

In order to assess the value of this approach to searching for nearest neighbors in parallel, parallel inverse distance weighting interpolation software was developed

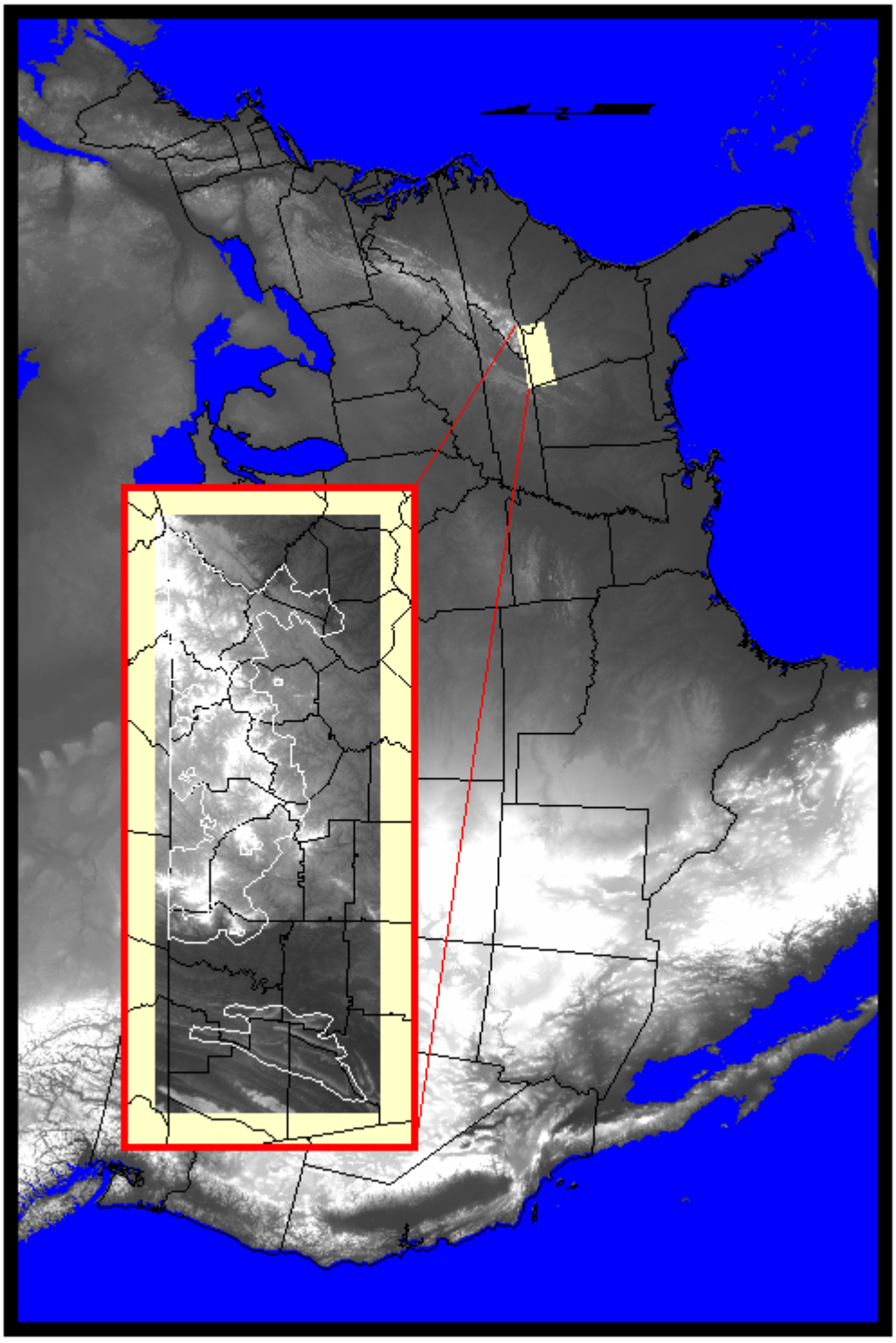


Figure 4.1. Study Area.

which applied this methodology. This software (*parallel.c*, Appendix E) was developed in C (again with the gcc compiler, version 2.95.2 on the Solaris 2.6 platform) using the MPICH 1.2.0 library. MPICH is a portable, freely available implementation of MPI which can be used to provide parallel processing capabilities to clusters of networked workstations and to multiprocessor machines. In order to optimize speed and to facilitate direct-to-disk writing for grid data produced by the software, the Points and Grid structures defined in the *geodata* module were not explicitly used. Rather the organization implied by these structures was implicitly exploited (that is, a raster or partial raster may be found directly in the grid file on disk by applying its expected location in memory in the hypothetical Grid structure).

As with many SIMD parallel applications, the *parallel* software was designed as a client/server application with a single processor acting to receive and write to disk partial rasters interpolated by the remaining processors (Figure 4.2). The remaining processors were arranged in a linear array whereby a single processor may only communicate with the processors above and below it (in addition to the server listening for partial rasters). Each client processor was responsible for loading a portion of data, searching for nearest neighbors and performing interpolation for a partial raster; partial rasters were sent back to the server to be written to disk as they were interpolated. Figure 4.3 illustrates the division of labor of partial rasters among processors.

Because elevation values sampled from the Digital Elevation Model are distributed throughout the plane, localization of workload in a brute force nearest neighbor search is difficult because each cell must compute the distance to all other points in order to absolutely determine the set of nearest neighbors. The net result in a parallel system is a highly expensive set of communications back and forth between each of the processors to completely share data. A more intelligent method to

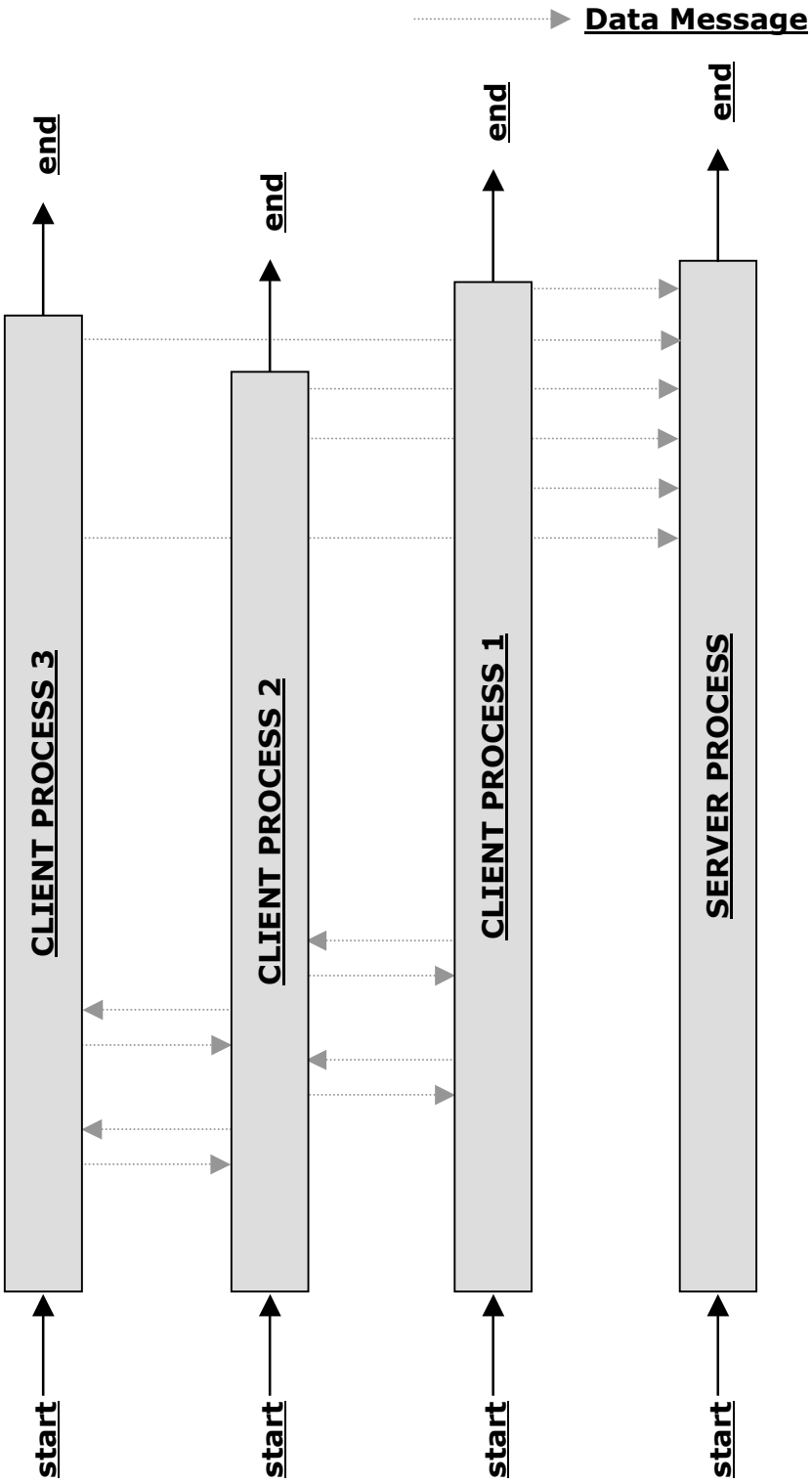


Figure 4.2. *Parallel* Software Architecture.

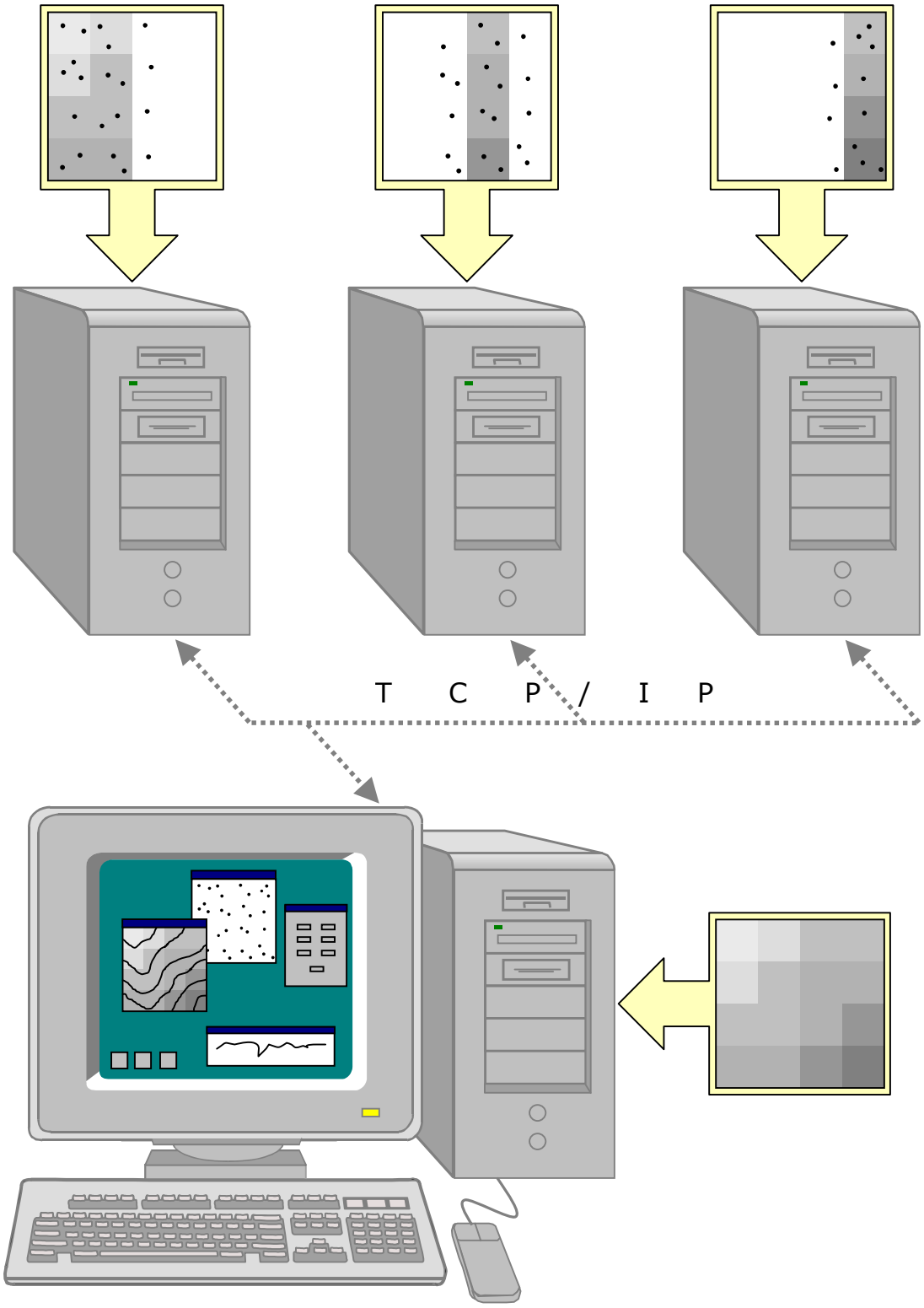


Figure 4.3. Division of Labor Among Partial Rasters.

searching for nearest neighbors was developed by Friedman *et al*. (1975) and adapted in the previous chapter to optimize parallel nearest neighbor searching. In the method by Friedman *et al*. (1975), the maximum number of points which must be searched in order to absolutely determine nearest neighbors may determined analytically as a

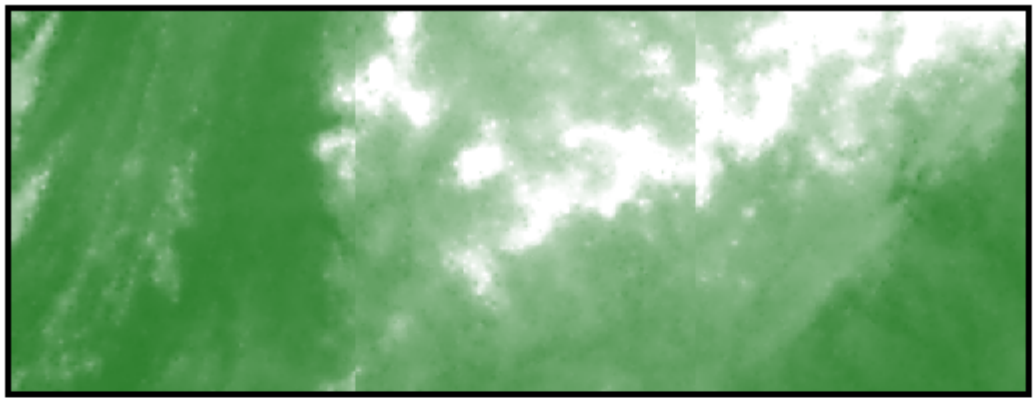
function of the number of points to be searched, the dimensionality of the data space and the number of nearest neighbors to be determined. Using this information, data may be more intelligently arranged so that (in the parallel case) the number of these expensive communications may be minimized. The Friedman *et al*. (1975) algorithm requires that all of the points be sorted along one dimension of the data space and is then able to search within that sorted data space. In the parallel implementation of this algorithm, the sorted data may be partitioned among the client processors. For example, with 3 clients and 10,000 points, subsets of 3,334 points may be located in memory on processors 1 and 2 with a subset of 3,332 points located in memory on processor 3. As indicated in the previous chapter, however, there is a problem with the straightforward implementation of this approach.

Interpolation for grid cell centers in the interior of the subset of points will proceed as expected and will produce the desired results. Interpolation for grid cell centers located near the edge of the subset of points, however, will not generate the expected results. This is due to the fact that points which should be within the domain of the nearest neighbor search are actually located in memory on adjacent processors. Because the boundary conditions of the nearest neighbor search assert that if the boundary of the dataset is reached with a fully determined set of – correct or incorrect – nearest neighbors, searching stops and that set of determined nearest neighbors is used as the definitive set of nearest neighbors. If the set of input points is "truncated" at the boundaries of the subset, then zones will occur near these boundaries in which results are incorrect. For sufficiently fine grid resolutions, these

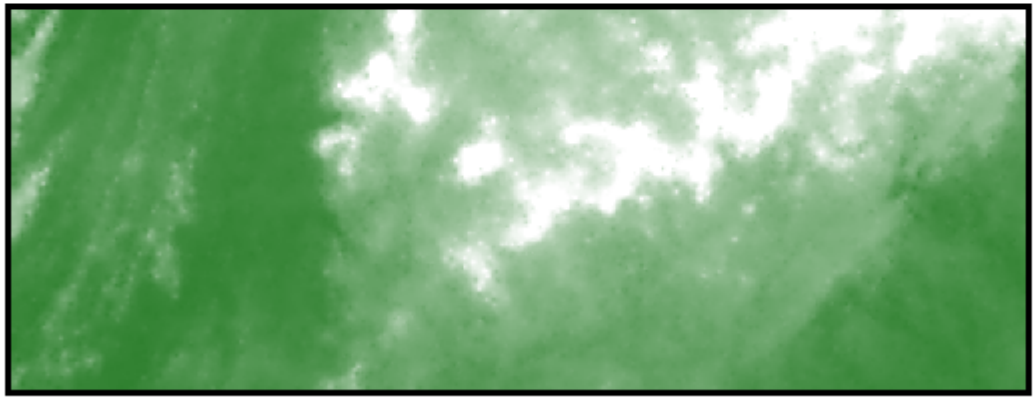
zones may not even be visible at some scales. For coarser grid resolutions, these zones may be more obvious. Figure 4.4 illustrates a problematic interpolation and the corrected interpolation. In order for the correct results to be generated at the boundaries of data some mechanism must be implemented to share data from adjacent processors so that grid cells at the boundary of the input data will have the omplete domain of input points available to them.

A simplistic first approach might be simply to request a point from an adjacent processor when it is needed. A recollection that interprocess communications are expensive operations reveals that this is not an optimal solution. There are two advantages to employing the Friedman *et al*. (1975) algorithm for parallelization of the nearest neighbor search. The first of these has already been applied; that is, the partitioning of the input points into a linearly ordered set which may be distributed across a linear array of parallel processors. The second advantage, however, has not been applied. Because this algorithm not only partitions data but also computes the maximum number of points which must be searched, a determination of how much data from adjacent processors will be required in order to correctly interpolate at the boundaries may be made. Using this information, the additional data which will be required may be obtained from adjacent processors in bulk a single time. Each processor will then contain sufficient data points to correctly find nearest neighbors (and thus interpolate) each cell which is assigned to it without need of further interprocess communications.

Extension of the parallel nearest neighbor search algorithm developed in the previous chapter to interpolation is straightforward and requires only a few modifications. Figure 4.5 shows the extension of the parallel nearest neighbor framework to inverse distance weighting interpolation. The only additional functionality which is required over the parallel nearest neighbor search is the mechanism for performing the actual interpolation and for storing interpolated rasters



Problematic Interpolation



Correct Interpolation

Figure 4.4. Problematic and Correct Interpolation Surfaces.

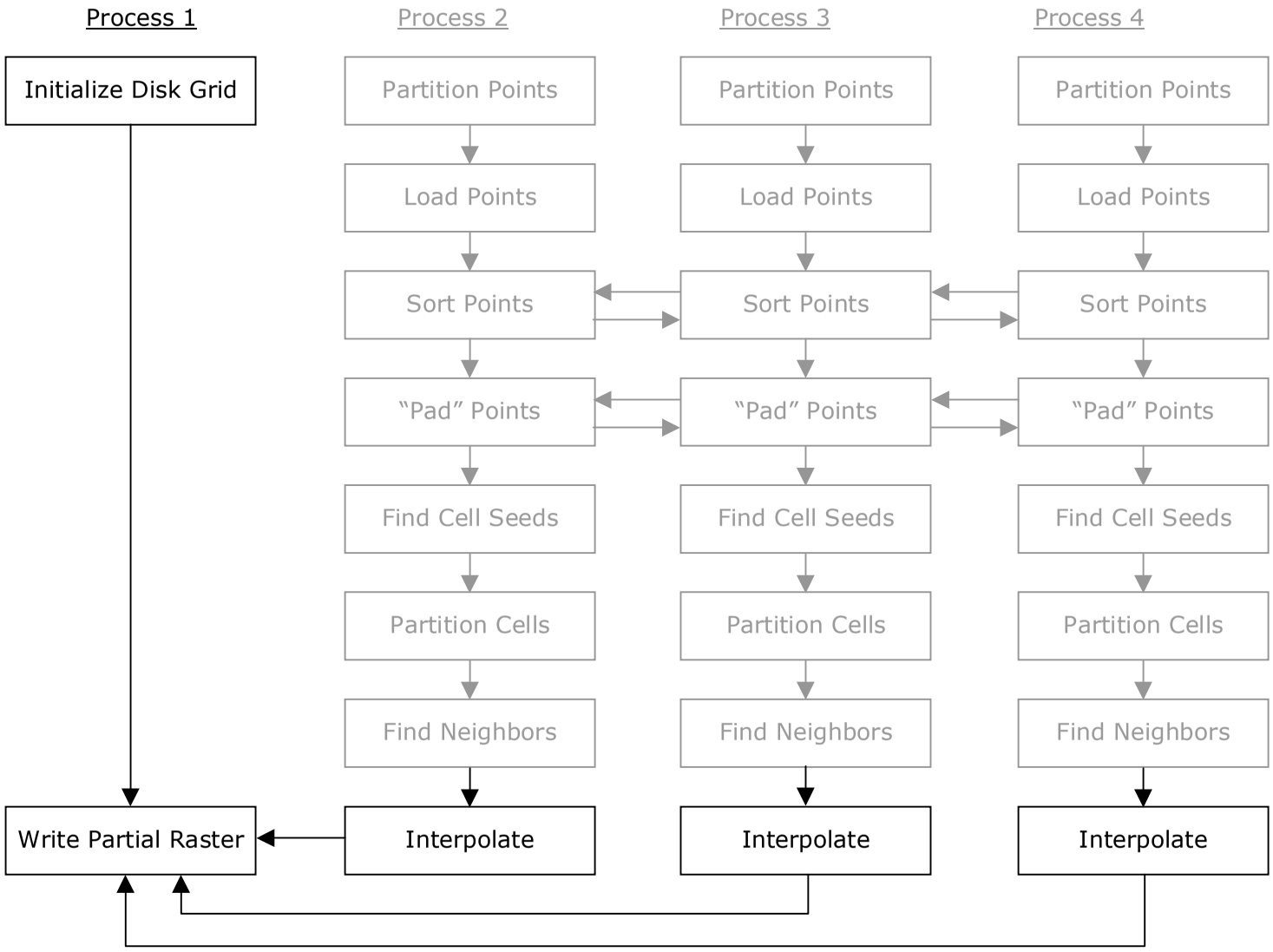


Figure 4.5. Extended Framework for Parallel Nearest Neighbor.

in a grid. In the case of the inverse distance weighting, the mechanism for performing the interpolation is minimal; in other interpolation routines such as kriging or in a geographic projection, this mechanism may be more sophisticated. However, the interpolation "box" may be viewed as a plug-in which is replacable by other techniques. Handling I/O to the grid file is more sophisticated in a system which does ot explicitly have parallel I/O capabilities. In such a system, either the interpolating processes must synchronize disk writes (to make sure that there are not multiple open file handles to the same file simultaneously) or they must communicate their results to a single processor which handles all of the file writes. The *parallel* software takes the second approach and dedicates a processor to grid file output.

The research initially done by Friedman *et al*. determined that the maximum number of points necessary for nearest neighbor interpolation could be determined as a function of (among other things) number of input points. Because this work did not deal with interpolation, no assessment was made of the variation that might be inherent as a function of number of grid cells. In order to establish a model for determining the number of required input points with varying numbers of grid cells, a sequential version of the *parallel* software was produced which captured the number of distance comparisons required per cell and averaged for each run by the number of cells. This process was executed for numbers of input points from 100,000 to 1,000,00 in multiples of 100,000 and with numbers of output grid cells from approximately 140,000 to 1,250,000 (corresponding to cell sizes decreasing from 360 m to 120 m). Figure 4.6 shows the results of this process and verifies that the mean distance computations (and hence the maximum number of required search points per cell) does not vary with number of grid cells.

To facilitate comparison, the same set of conditions that were applied earlier to determine computation time using Arc/Info on a single processor were applied to the *parallel* software. That is, number of input points again varied from 100,000 to 1,000,000 by 100,000 and number of grid cells varied from approximately 140,000 to 1,250,000. Executions of the *parallel* software ran all four processors of a four

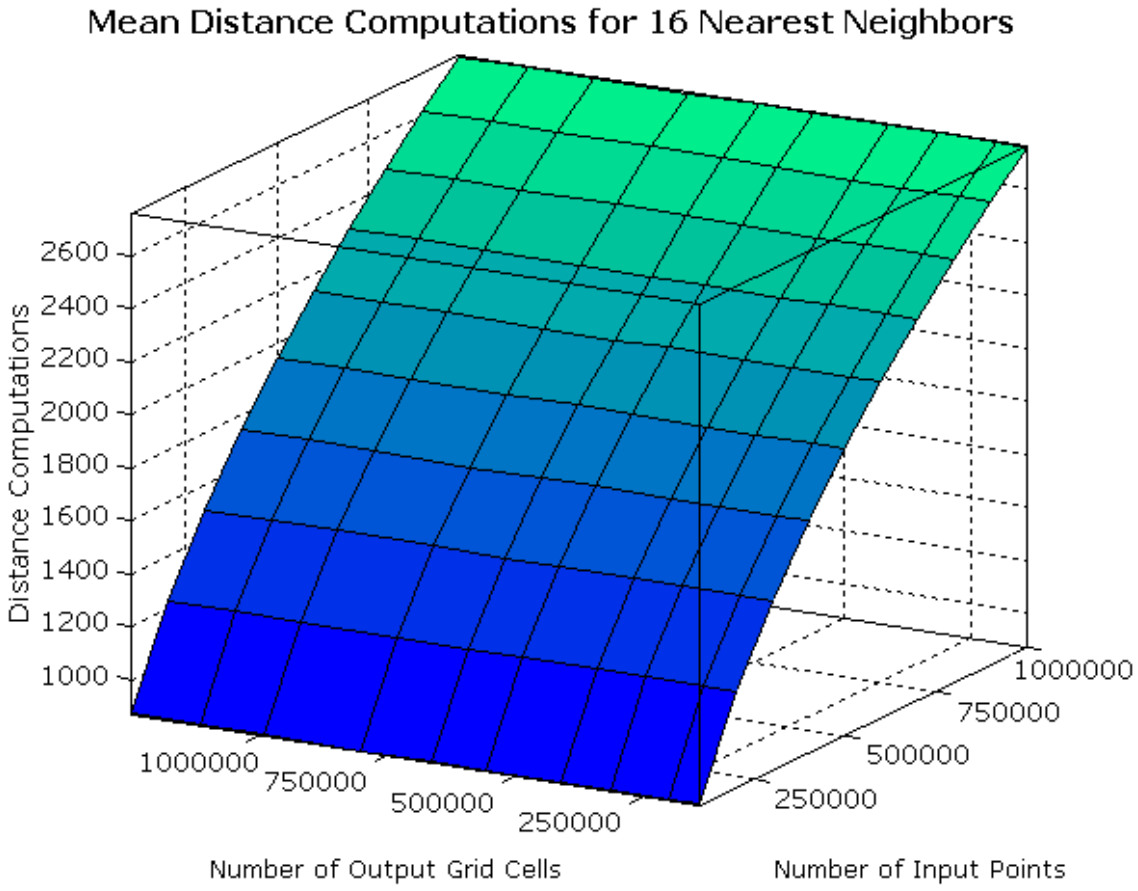


Figure 4.6. Mean Distance Computations.

processor Sun Enterprise 450 server (400 MHz per processor). For each execution, wall-clock run times were captured. Results will be discussed in the next chapter.

CHAPTER 5

RESULTS AND DISCUSSION

Inverse distance weighting interpolations have been performed both for a sequential architecture (Arc/Info) and for a parallel architecture via MPI. For both sets of 100 runs, numbers of input points varied from 100,000 to 1,000,000 in multiples of 100,000 and numbers of grid cells varied from approximately 140,000 to 1,250,000 (corresponding to cell sizes from decreasing from 360 m to 120 m). This inverse distance weighting interpolation was performed using Euclidean distance for 16 nearest neighbors. For the sequential Arc/Info based version, the interpolation was performed on a single processor of a four 400 MHz processor Sun Enterprise 450 server while the parallel version used all four processors networked together via MPI. Results were timed for each set of 100 runs.

As indicated previously, times for sequential runs ranged from 2.28 minutes for 100,000 points and 139,840 grid cells to 73.07 minutes for 1,000,000 points and 1,253,536 grid cells. The trend appears to be an exponential increase in sequential processing times as the number of points increases and a linear increase in sequential processing times as the number of grid cells increases (although the degree of linearity appears to vary with the number of input points). The overall effect is an apparent exponential increase as the number of grid cells and input points increase. Figure 5.1 is a parametric surface defined by {number of input points, number of output grid cells, sequential processing time} and illustrates these general trends for sequential interpolation processing times. Table 5.1 summarizes individual time values for each sequential run.

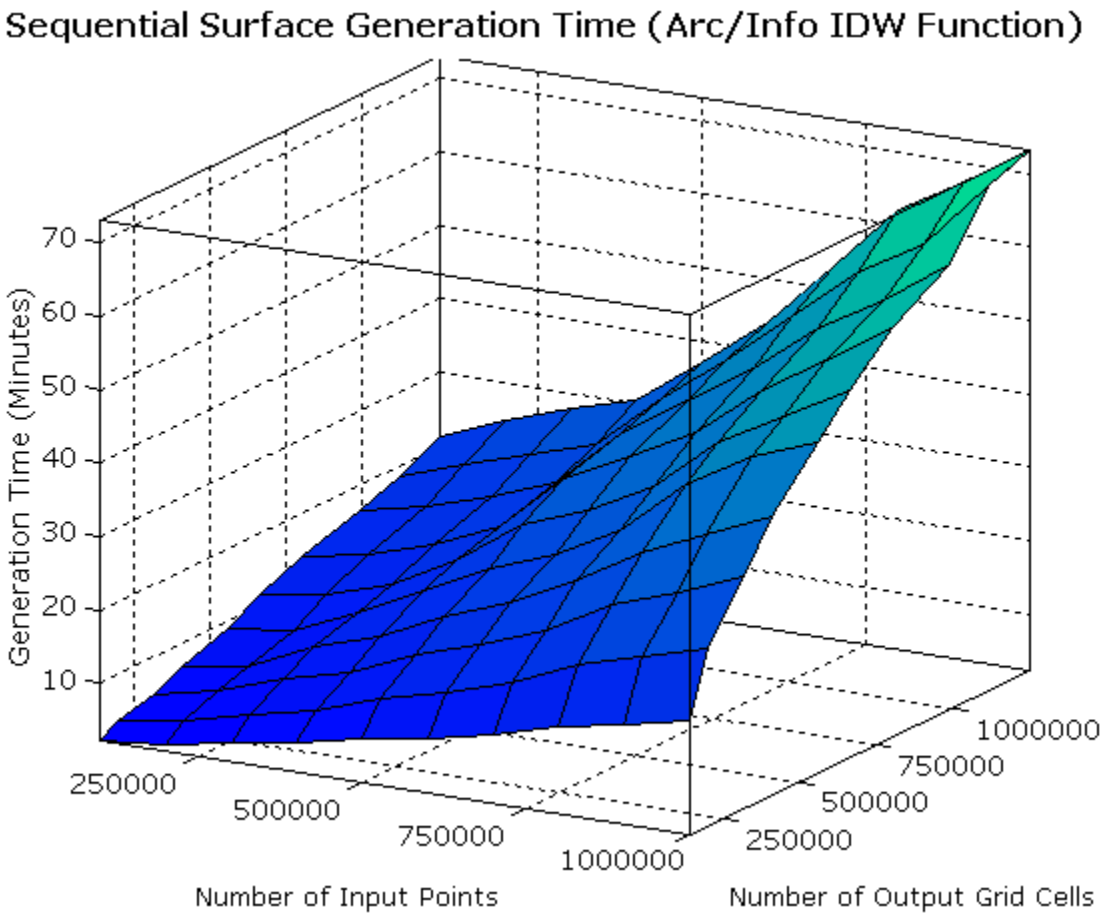


Figure 5.1. Sequential Processing Times.

POINTS / CELLS	100000	200000	300000	400000	500000	600000	700000	800000	900000	1000000
139840	2.28	3.18	4.78	6.10	8.15	9.60	11.53	14.12	15.90	17.75
201204	3.80	5.18	7.42	9.48	12.45	14.63	17.45	21.25	23.82	26.52
314640	5.00	6.87	10.02	12.22	16.22	18.93	22.38	27.15	30.40	33.82
410548	6.90	8.22	12.53	15.37	19.33	22.72	27.05	32.48	36.28	40.25
557685	9.12	10.33	14.23	18.50	22.98	26.20	31.15	38.73	43.85	47.20
663825	11.52	12.90	15.95	20.57	26.98	30.25	34.93	42.38	47.37	52.25
803358	13.97	15.60	18.38	22.40	30.30	34.68	39.48	46.67	52.20	57.95
991440	16.38	18.50	21.32	24.50	32.78	38.53	44.30	51.22	56.67	62.80
1119300	18.62	21.48	24.53	27.30	34.83	41.55	49.00	56.47	61.73	71.05
1253536	21.20	24.85	28.03	30.60	37.10	44.03	53.02	62.17	67.37	73.07

Table 5.1. Sequential Processing Times in Minutes.

Times for parallel runs ranged from 0.48 minutes for 100,000 points and 139,840 grid cells to 12.08 minutes for 1,000,000 points and 1,253,536 grid cells. Unlike sequential processing times, parallel processing times appear to increase linearly for both variables and linearly overall (although more steeply linear as both number of input points and number of output grid cells increases). Figure 5.2 is a parametric surface defined by {number of input points, number of output grid cells, parallel processing time} which illustrates these trends for parallel interpolation processing time. The parametric surface of Figure 5.1 is overlain as a mesh as well in order to facilitate direct comparison between parallel and sequential run times. Table 5.2 summarizes the individual run times for each parallel run.

A particular characteristic of the parallel processing run times worth noting is that (at least by visual inspection) parallel processing appears to be much more scalable than sequential processing. Because of the linearity or near-linearity of processing times for the parallel implementation, larger sets of either input or output data should not cause processing times to increase so dramatically is to become infeasible. By comparison, processing times for sequential interpolation appear to increase so quickly as to be impractical for data sets much larger than the problem domain of this work. Bearing in mind as well that for parallel implementations, as processing times become impractical, more processors may be added to the cluster to increase processor power and commensurately decrease processing time. This in turn leads to increased scalability as processing times may be manipulated by adding or subtracting processors from the parallel cluster. It should be noted as well that more complex routines such as kriging or projection may alter these processing times, but the inverse distance weighting interpolation requires so little overhead that it is a good mechanism for measuring nearest neighbor search times for a complete dataset (or at least a reasonable approximation).

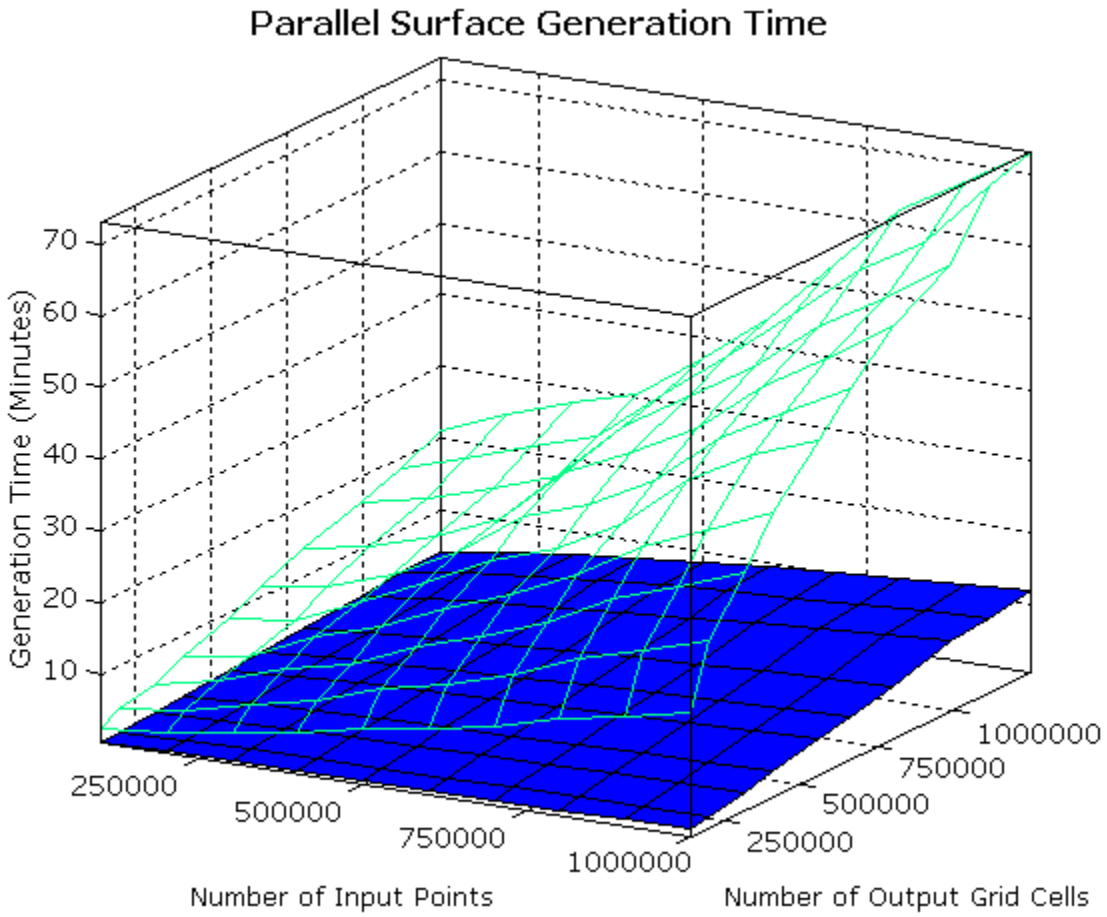


Figure 5.2. Parallel Processing Times.

POINTS / CELLS	100000	200000	300000	400000	500000	600000	700000	800000	900000	1000000
139840	0.48	0.65	0.82	0.95	1.05	1.12	1.20	1.28	1.40	1.50
201204	0.67	0.90	1.13	1.28	1.43	1.58	1.72	1.83	1.98	2.10
314640	1.02	1.42	1.72	1.97	2.20	2.43	2.60	2.80	2.97	3.17
410548	1.35	1.85	2.27	2.57	2.88	3.12	3.38	3.63	3.85	4.08
557685	1.82	2.52	3.03	3.47	3.87	4.30	4.58	4.88	5.18	5.50
663825	2.15	3.03	3.57	4.12	4.55	5.07	5.45	5.82	6.17	6.53
803358	2.63	3.57	4.37	4.98	5.57	6.12	6.53	6.98	7.38	7.87
991440	3.20	4.43	5.35	6.12	6.82	7.63	8.48	9.10	9.70	10.23
1119300	3.90	5.37	6.53	7.40	8.17	8.80	9.30	9.85	10.35	10.82
1253536	4.10	5.60	6.73	7.68	8.58	9.45	10.02	10.70	11.40	12.08

Table 5.2. Parallel Processing Times in Minutes.

Perhaps as important as the absolute parallel processing time is the relative improvement between sequential and parallel run times; that is the proportion of sequential time required for parallel processing. Proportions of processing times ranged from 0.2711 for 400,000 input points and 1,119,300 grid cells at the upper extreme to 0.0792 for 1,000,000 sample points and 201,204 grid cells at the lower extreme. The proportional processing time for the maximum values of 1,000,000 input points and 1,253,536 grid cells was 0.1654. The mean proportional processing time was 0.1695 with a standard deviation of 0.0484. In general, proportional processing times were highest for low numbers of points and then decreased commensurately as numbers of input points increased. This fact is key because it clearly illustrates the increasing value of this method as the number of input and output data increase. As with absolute parallel processing times, it is expected that additional processors would further decrease the proportional processing time. Figure 5.3 illustrates the sequential to parallel processing proportion.

There are as well some benefits to be gained which are not necessarily easy to directly observe. In particular, a challenge which often faces parallel processing systems is the idea of load balancing; that is, the process of insuring that no single processor waits inordinately for work to do. With the methodology presented in this work, the only time that a processor may be sitting idle is in the parallel merge-split sort. Because this sort is such a small part of the computation in this method (sorting 1,000,000 points in the x direction typically required approximately 3 seconds), such idling is completely acceptable. Once the sort has finished, however, there is no future point at which the processor sits idle; each processor knows its responsibility exactly and contains all of the data required to execute that responsibility.

The benefits bestowed by decreased processing times are meaningless if correct results are not produced. It is particularly important in parallel systems to verify that results are complete and correct in order to avoid the introduction of artifacts due to

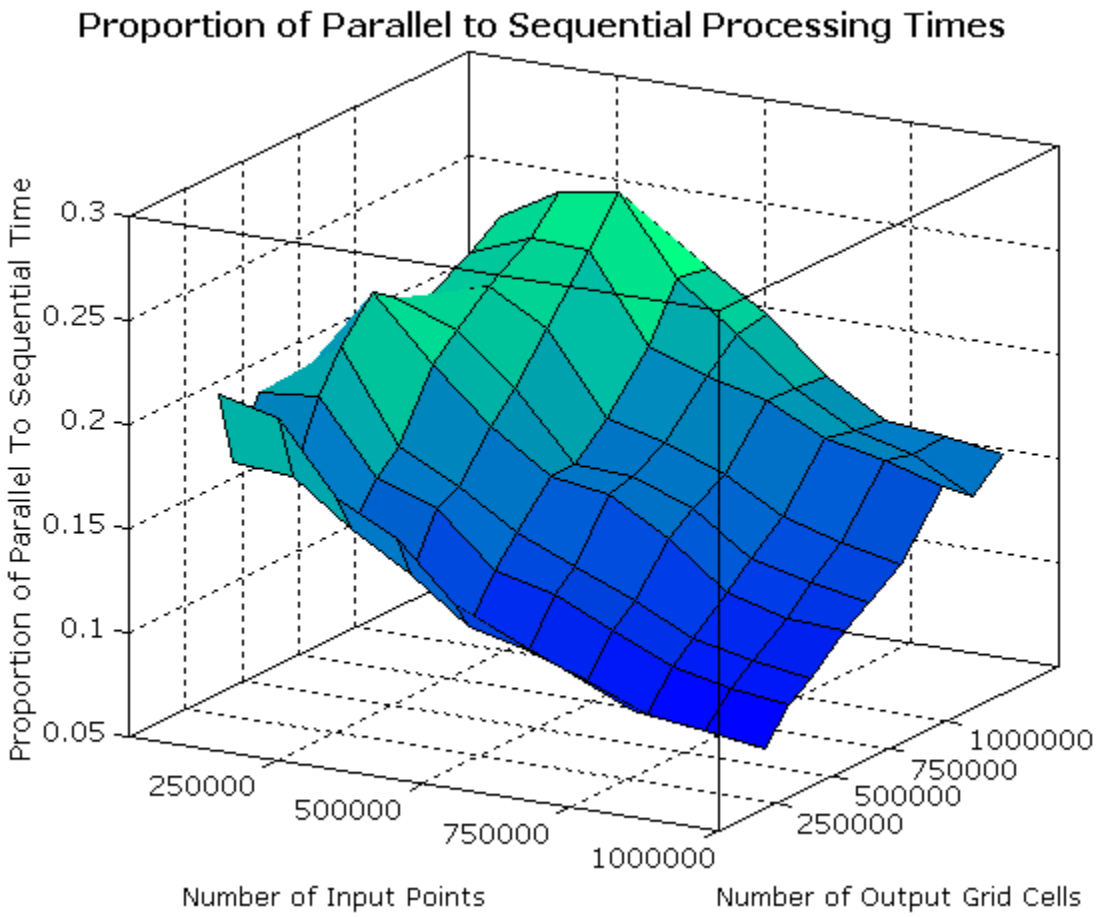


Figure 5.3. Proportion of Parallel to Sequential Processing Times.

POINTS	100000	200000	300000	400000	500000	600000	700000	800000	900000	1000000
CELLS										
139840	0.2117	0.2042	0.1707	0.1557	0.1288	0.1163	0.1040	0.0909	0.0881	0.0845
201204	0.1754	0.1736	0.1528	0.1353	0.1151	0.1082	0.0984	0.0863	0.0833	0.0792
314640	0.2033	0.2063	0.1714	0.1610	0.1357	0.1285	0.1162	0.1031	0.0976	0.0936
410548	0.1957	0.2252	0.1809	0.1670	0.1491	0.1372	0.1251	0.1119	0.1061	0.1014
557685	0.1993	0.2435	0.2131	0.1874	0.1682	0.1641	0.1471	0.1261	0.1182	0.1165
663825	0.1867	0.2351	0.2236	0.2002	0.1686	0.1675	0.1560	0.1372	0.1302	0.1250
803358	0.1885	0.2286	0.2375	0.2225	0.1837	0.1764	0.1655	0.1496	0.1414	0.1357
991440	0.1953	0.2396	0.2510	0.2497	0.2079	0.1981	0.1915	0.1777	0.1712	0.1630
1119300	0.2095	0.2498	0.2663	0.2711	0.2344	0.2118	0.1898	0.1744	0.1677	0.1522
1253536	0.1934	0.2254	0.2402	0.2511	0.2314	0.2146	0.1889	0.1721	0.1692	0.1654

Table 5.3. Proportion of Parallel to Sequential Processing Times.

the architecture of the system. Figures 5.4 and 5.5 show the original Digital Elevation Model (30m cellsize) and its corresponding shaded relief. Figures 5.6 and 5.7 illustrate a surface and its corresponding shaded relief produced by the *parallel* software for 1,000,000 points and 120m cellsize (corresponding to approximately 1,250,000 grid cells). Note that processing time using the IDW routine in Arc/Info required 73 minutes while the *parallel* software required only 12 minutes. Both sequential and parallel interpolations produced identical results. In order to verify results, the binary grid produced by the *parallel* software was converted first to GeoTIFF format using the *convert* software. This GeoTIFF was then imported into Arc/Info in grid format. Likewise, the binary points file containing the input data to the *parallel* software was converted to Shapefile format by the *convert* software. The Shapefile was then imported into Arc/Info in coverage format and used as input to the IDW function in Grid to interpolate the surface with the same parameters. The surface generated by *parallel* was then subtracted from the surface generated by IDW to produce an error surface whose largest values were less than ± 0.5; that is the error term between the two surfaces could be attributed to roundoff error because the GeoTIFF stores only whole numbers while the Arc/Info grid may store values in floating point format.

In order to evaluate scalability of the approach, interpolations were also performed using both Arc/Info and *parallel* for smaller numbers of input points and cellsizes. Even for very small datasets (1,000 points and 1,000 cells), parallel interpolation still took at most 50% of the time required for sequential interpolation – although in this case the difference was between 1 and 2 seconds. Likewise, for 10,000 input points and 10,000 cells, parallel processing required approximately 28% of the time required for sequential interpolation (2 and 7 seconds respectively). Although theoretically beneficial for these low numbers, practical application may demand higher absolute returns before the effort of parallelization is undertaken –

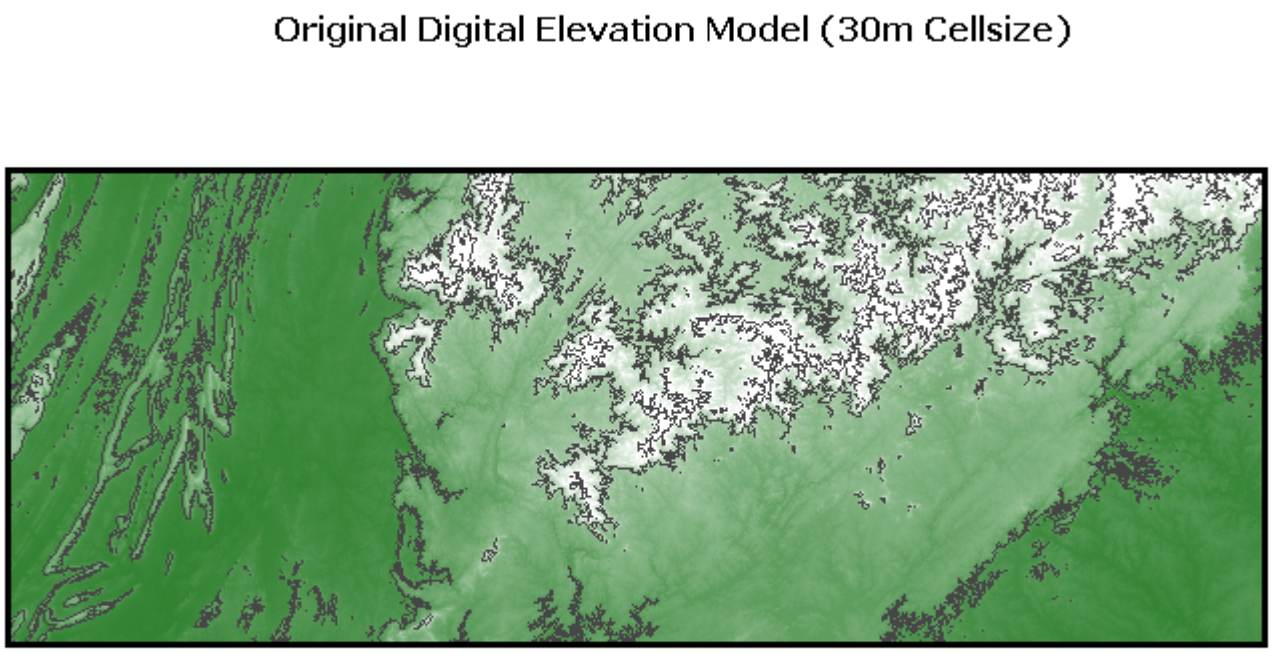


Figure 5.4. Original Digital Elevation Model.

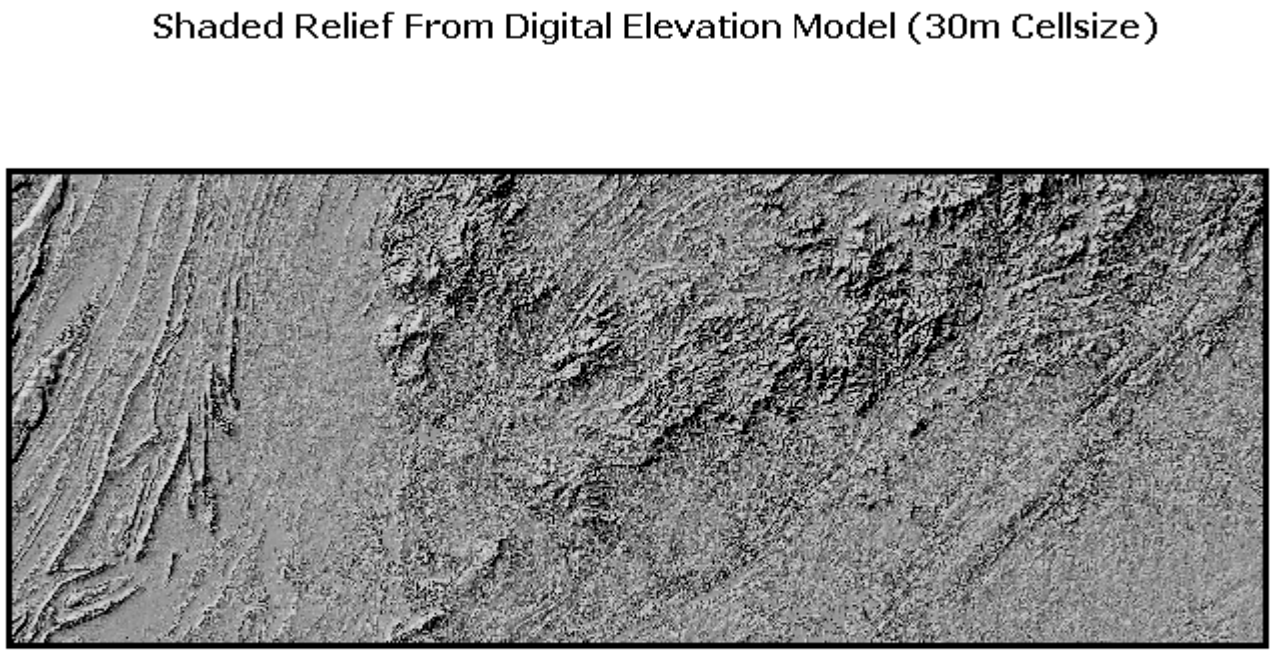


Figure 5.5. Shaded Relief From Digital Elevation Model.

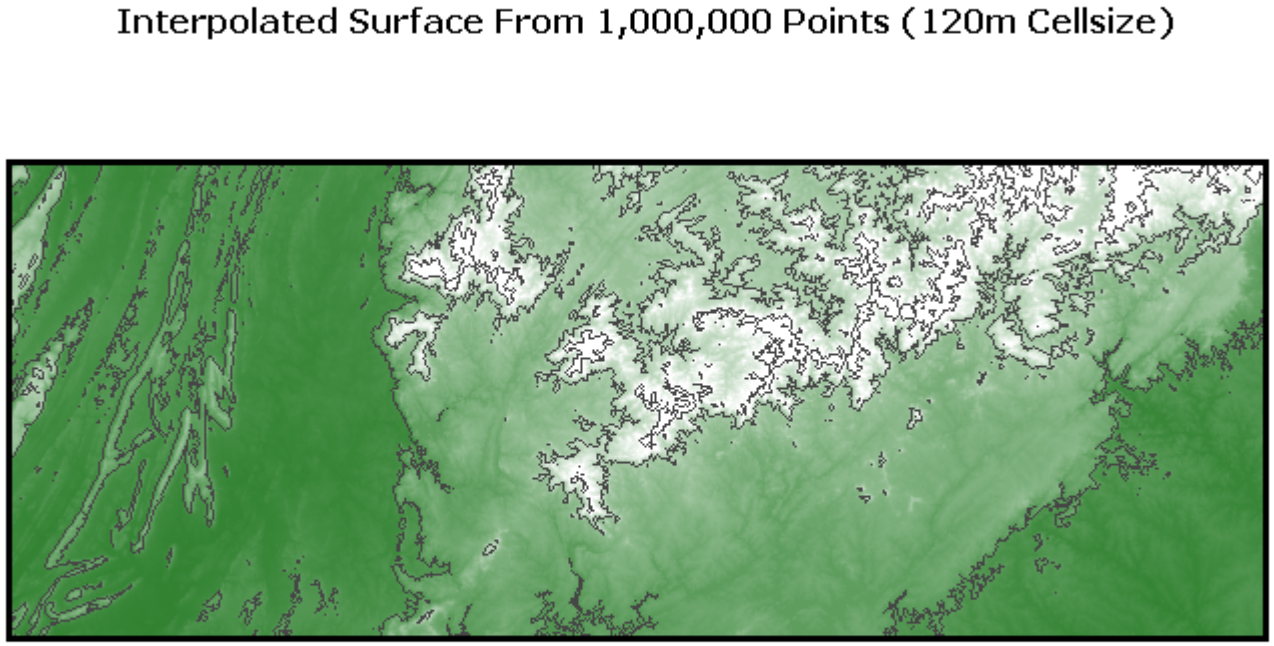


Figure 5.6. Interpolated Surface From *parallel* Software.

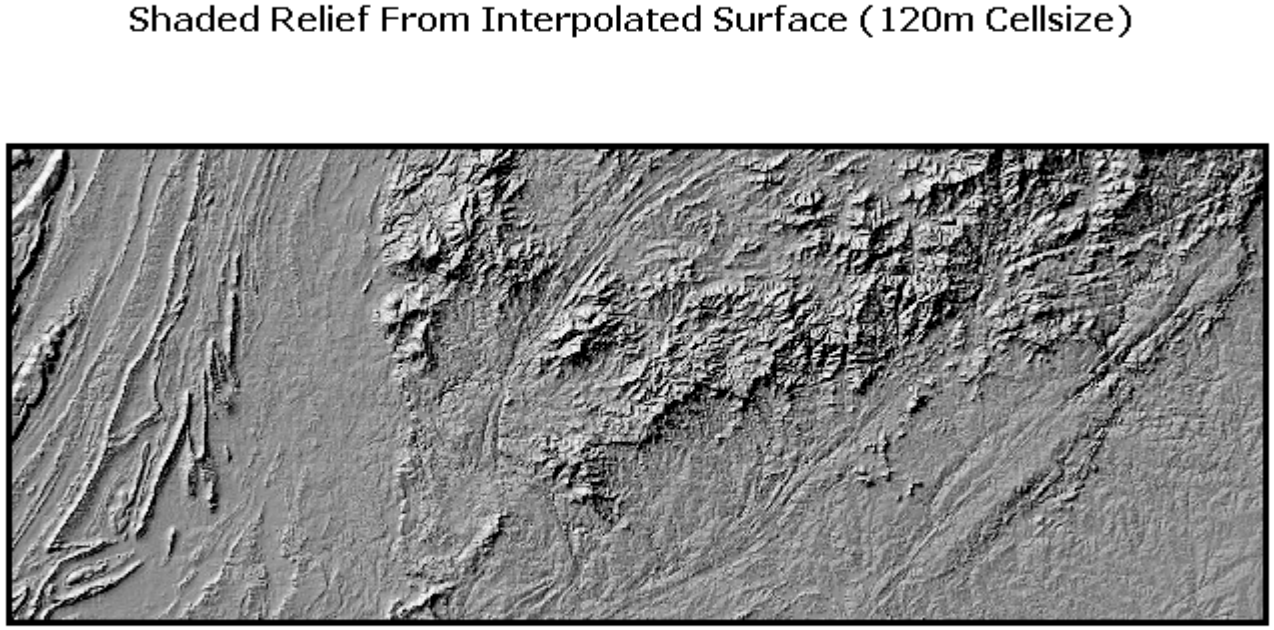


Figure 5.7. Shaded Relief From Interpolated Surface.

perhaps for interpolation times more than 2 minutes – corresponding to approximately 100,000 points and 100,000 cells. Although this work presents a methodology for parallelizing the nearest neighbor search, parallelization of other interpolation routines besides the Inverse Distance Weighting (for example, Kriging) will require additional parallel methodologies beyond the nearest neighbor search. An understanding of the costs and benefits of parallelization should be understood before this effort is undertaken.

CHAPTER 6

CONCLUSIONS

A method has been developed for parallelizing nearest neighbor search operations by exploiting a sequential algorithm for nearest neighbor searching presented by Friedman *et al*. This method has the properties not only that it explicitly parallelizes the nearest neighbor search problem but also that minimizes necessary interprocess communications while guaranteeing correct and complete solutions. Because the number of distance computations is bounded, the volume of data needed by each processor in addition to the data already resident in memory may be determined a priori and therefore may be localized on the processor. The net result is that subset data domains actually overlap in order to eliminate interprocess communications during the nearest neighbor phase of the operation. Indeed, the only required interprocess communications occur during the initial sort which is a precondition to the operation and during the swap of overlapping data to complete the set of input data before starting actual nearest neighbor search procedures.

In order to verify the efficacy of this approach, the relationship between number of input points, number of output grid cells and surface generation time have been considered for both a sequential implementation of the inverse distance weighting interpolation using the Arc/Info package and for a parallel implementation in C using MPI to achieve interprocess communications for 100 permutations of 10 linearly increasing input point sizes and 10 approximately linearly increasing number input grid cells. Parallel computations were performed on a four processor Sun Enterprise 450 server while sequential computations were performed on a single

processor on the same Sun Enterprise 450. Resultant run times showed that parallel processing times were significantly reduced from sequential processing times, with the mean case requiring approximately 16% of the sequential processing time to perform the computation in parallel. Moreover as input and output datasizes increase, the proportion of time required for the parallel case appears to decrease exponentially.

Although the body of literature addressing parallel GIS is growing, much of it focuses on parallel data structures and parallel vector operations (and particularly the parallel Delaunay Triangulation and its counterpart in engineering, the Finite Element Method). Much more limited is the research on parallel raster operations and in particular work dealing with nearest neighbor searches. Although it has been demonstrated that the Delaunay Triangulation may be applied to efficient solution of nearest neighbor problems, it suffers from the drawback that data cannot be efficiently partitioned among processors (to reduce the overall amount of data which must be held in the memory of a single processor at one time) without requiring interprocess communication. For large scale problems where the amount of data exceeds the amount of available memory on a processor, it is desirable to parallelize the nearest neighbor search in such a way that the sum of memory on several processors is sufficient to solve the problem, even though the memory on a single component processor may not be sufficient. The Delaunay Triangulation does not explicitly deal with this case. This method readily deals with this problem and is scalable.

Further Applications

The inverse distance weighting interpolation used in this research is known to produce generally inferior results to other interpolation methods such as kriging. The purpose of the use of the inverse distance weighting routine in this work was to demonstrate the application of this method for parallel nearest neighbor searching

without undue overhead attributable to the interpolation method itself. For real-world applications, however, it would be preferable to be able to apply this work to a more robust interpolation method. In particular, kriging is widely recognized to produce superior results at the expense of very long processing times. This method may be used as the basis for parallelizing the kriging interpolation. There are two principal parts to the kriging interpolation: computation of the variogram and the actual search for nearest neighbors and application of the interpolation. Computation of the variogram in general comprises a small part of the overall computation of the kriging method. This work may serve as the basis for parallelizing the second component of the kriging interpolation; that is, nearest neighbor searching may occur in parallel. The remainder of the work in parallelizing this operation then is addressing the application of the variance data to perform the interpolation. Although this certainly is a substantial problem, the necessary work for the overall problem is reduced.

Another common application of nearest neighbor searching deals with the geoprocessing operation of projection where a grid is taken from one coordinate system to another. In general, projection changes the shape (or at least the orientation of the grid), and determination of values for the new grid cells must be carried out through nearest neighbor, bilinear or bicubic interpolation. This method may be readily applied to the problem to find the 1, 4 or 16 nearest neighbors to perform the interpolation. In the case of projection, the input points to be searched will actually be the cell centers of the old grid in their new projected positions. Parallelization of projection then is straightforward and also requires two steps; the first step will be to apply the mathematical transformation which defines the projection to the cell centers of the old grid to obtain new coordinate values. The second step will be simply execution of this method on the set of projected cell centers. For nearest neighbor interpolation, the remaining work is minimal – simply

the nearest neighbor to the new cell center must be found and its value used as the new cell value. For bilinear and bicubic interpolation, the four or sixteen nearest neighbors must be found (as they may be with the approach presented in this work) and then averaged – a process which requires even less overhead than the inverse distance weighted interpolation. Parallelization of projection, then, is completely straightfoward.

Issues and Future Directions

Although demonstration of the inverse distance weighting interpolation was executed on a single multiprocessor server, this method is extensible to a cluster of powerful workstations. In fact, no changes to the code presented in Appendix E are necessary to implement the software on a distributed parallel system (although changes to the cluster configuration in the MPI parameter files are required). Further work in this regard should evaluate the applicability of distributed parallel processing in order to assess the additional overhead in such a solution due to message passing across a network. Additionally, this methodology was tested only for 16 nearest neighbors – a "worst case" scenario (typically not more than 16 nearest neighbors are used for interpolation). A future development of this work might also evaluate the applicability and benefit of this algorithm for fewer neighbors – such as 1 or 4 – or possibly more neighbors since the overhead of finding the neighbors may be so substantially reduced.

In addition to these improvements, it may also be desirable to more rigorously evaluate benefits through statistical analysis of the results obtained with a larger dataset. This is not a slight undertaking, due largely in part to the time that will be required to collect sequential processing times for comparison. It would also be valuable to assess improvements for even larger sets of input points and output grid cells.

A final future direction may well be to implement this approach to assess its actual benefit to kriging and projection. Although parallelization of such methods is easily extrapolated, determination of the actual benefit incurred by parallelization is key, particularly for the development of a set of robust parallel techniques for geostatistics and geoprocessing.

REFERENCES

Akl, S. 1989. *The Design and Analysis of Parallel Algorithms.* Englewood Cliffs, New Jersey: Prentice Hall.

Akl, S. 1985. *Parallel Sorting Algorithms.* New York: Harcourt Brace Jovanovich.

Armstrong, M., and Densham, P. 1992. Domain Decomposition for Parallel Processing of Spatial Problems. *Computers, Environment and Urban Systems*. 16: 497-513.

Armstrong, M. 1994. GIS and High Performance Computing. *In Proceedings of GIS/LIS'94, pp. 4-13.*

Armstrong, M. 1995. Is There a Role for High Performance Computing in GIS? *URISA Journal*. 7(2): 7-10.

Armstrong, M., and Marciano, R. 1994. Local Interpolation Using a Distributed Parallel Supercomputer. *International Journal of Geographical Information Systems.* 10(6): 713-729.

Bailey, T., and Gatrell, C. 1995. *Interactive Spatial Data Analysis.* New York: John Wiley & Sons.

Bernhardsen, T. 1992. *Geographic Information Systems*. Arendal, Norway: Viak IT.

Berry, B. 1964. Approaches to Spatial Analysis: A Regional Synthesis. *Annals of the Association of American Geographers*. 54: 2-11.

Bertsekas, D. and Tsitsiklis, J. 1989. *Parallel and Distributed Computation: Numerical Methods*. Englewood Cliffs, New Jersey: Prentice Hall.

Burrough, P. 1986. *Principles of Geographical Information Systems for Land Resources Assessment*. New York, NY: Oxford University Press.

Clarke, K. 1990. *Analytical and Computer Cartography.* Englewood Cliffs, New Jersey: Prentice Hall.

Clematis, A., Falcidieno, B., and Spagnuolo, M. 1996. Parallel Processing on Heterogeneous Networks for GIS Applications. *International Journal of Geographical Information Systems.* 10(6): 747-767.

Dangermond, J. and Morehouse, S. 1987. Trends in Hardware for Geographic Information Systems. In *Proceedings of Auto-Carto 8*. Bethesda: American Congress for Surveying and Mapping. 380 –385.

Densham, T., and Armstrong, M. 1998. Spatial Analysis. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 387-413.

Deelman, E. and Szymanski, B. 1998. Dynamic Load Balancing in Parallel Discrete Event Simulation for Spatially Explicit Problems. *In Proceedings of the Workshop on Parallel Distributed Simulation PADS, IEEE Computer Society.* pp. 46-53.

Ding, Y., and Densham, P. 1996. Spatial Strategies for Parallel Spatial Modeling. *International Journal of Geographical Information Systems.* 10(6): 669-698.

Dowers, S., Gittings, B., Sloan, T. and Waugh, T. 1991. Analysis of GIS Performance on Parallel Architectures and Workstation-Server Systems. *In Proceedings of GIS/LIS'91,*pp. 555-561.

Dowers, S. 1998. Data Models, Representation and Interchange Standards. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 103-138.

Dowers, S., Mineter, M. and Healey, R. 1998a. A Modular Approach to Parallel GIS Algorithm Design. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 139-150.

Dowers, S., Mineter, M. and Healey, R. 1998b. Issues in the Design of Parallel Algorithms. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 89-101.

Flynn, M. 1966. Very High Speed Computing Systems. *Proceedings of the IEEE.* 54(12): 1901-1909.

Flynn, M. 1972. Some Computer Organizations and their Effectiveness. *IEEE Transactions on Computers.* C-21: 948-960.

Friedman, J., Baskett, F., and Shustek, L. 1975. An Algorithm for Finding Nearest Neighbors. *IEEE Transactions on Computers.* 10: 1000-1006.

Goodchild, M. 1992. Geographical Information Science. *International Journal of Geographical Information Systems*. 6(1): 31-45.

Green, P., and Sibson, R. 1977. Computing Dirichlet Tesselations in the Plane. *The Computer Journal.* 21(2):168-173.

Hambrusch, S. and Khokar, A. 1997. Maintaining Spatial Data Sets in Distributed-Memory Machines. *In Proceedings of the International Parallel Processing Symposium IPPS, IEEE.* pp. 702-707.

Harding, T., Healey, R., Hopkins, S. and Dowers, S. 1998. Vector Polygon Overlay. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 265-310.

Healey, R., Gittings, B., Dowers, S., and Mineter, M. 1998. Introduction. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings, and Mike Mineter, Eds. London: Taylor and Francis. 1-9.

Healey, R., Dowers, S., Gittings, B., and Tranter, M. 1998. Parallel Database Management Systems for GIS. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 333-349.

Hodgson, M. 1989. Searching Methods for Rapid Grid Interpolation. *Professional Geographer.* 41(1): 51-61.

Lam, N. 1983. Spatial Interpolation Methods: A Review. *The American Cartographer.* 10(2): 129-149.

Lee, D. 1982. k-Nearest Neighbor Voronoi Diagrams in the Plane. *IEEE Transactions on Computers.* C-31(6): 478-487.

Li, B. 1992. Opportunities and Challenges of Parallel Processing in Spatial Data Analysis: Initial Experiments with Data Parallel Map Analysis. *In Proceedings of GIS/LIS'92,* pp. 445-458.

Magillo, P., and Puppo, E. 1998. Algorithms for Parallel Terrain Modelling and Visualisation. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 351-386.

Mineter, M. 1998a. Creation of Vector-Topological Data Structures. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 179-214.

Mineter, M. 1998b. Partitioning Raster Data. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 215-230.

Mineter, M. 1998c. Raster-to-Vector Conversion. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 253-264.

Mineter, M., Healey, R., Gittings, B., and Dowers. S. 1998a. Towards Parallel Libraries for Geographical Algorithms." *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 441-445.

Mineter, M., Wilkinson, G., Dowers, S., and Healey, R. 1998b. Implementation Case Study: Generalisation of Raster Data. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 311-330.

Mower, J. 1996. Developing Parallel Procedures for Line Simplification. *International Journal of Geographic Information Systems.* 10(6): 699-712.

Mower, J. 1992. Building a GIS for Parallel Computing Environments. *In Proceedings of the 5th International Symposium on Spatial Data Handling,*pp. 219-229.

Press, W., Flannery, B., Teukolsky, S., and Vetterling, W. 1988. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press: New York.

Roche, S., and Gittings, B. 1996. Parallel Polygon Line Shading: The Quest for More Computational Powerfrom an Existing GIS Algorithm. *International Journal of Geographic Information Systems*. 10(6): 731-746.

Sawyer, M. 1998a. The Development of Hardware for Parallel Processing. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 13-31.

Sawyer, M. 1998b. The Software Environment and Standardisation Initiatives. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 33-57.

Shekar, S., Ravada, S., Kumar, V., Chubb, D., and Turner, G. 1998. Declustering and Load-Balancing Methods for Parallelizing Geographic Information Systems. *IEEE Transactions on Knowledge and Data Engineering.* 10(4): 632-655.

Shekar, S., Ravada, S., Kumar, V., Chubb, D., and Turner, G. 1996. Parallelizing a GIS on a Shared Address Space Architecture. *Computer*. 29(12): 42-48.

Sloan, T., and Dowers, S. Parallel Vector Data Input. 1998. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 151-178.

Sloan, T. 1998. Vector-to-Raster Conversion. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 233-252.

Trewin, S. 1998. High-Level Support for Parallel Programming. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis. 59-86.

Verts, W. and Thomson, C. 1988. Parallel Architectures for Geographic Information Systems. *In Technical Papers of the ACSM-ASPRS Annual Convention*, pp. 101-107.

Wilkinson, G. 1998. Parallel Processing Approaches in Remotely Sensed Image Analysis. *Parallel Processing Algorithms for GIS.* Richard Healey, Steve Dowers, Bruce Gittings and Mike Mineter, Eds. London: Taylor and Francis, 1998. 415-437.

Xavier, C., and Iyengar, S. 1998. *Introduction to Parallel Algorithms*. New York: John Wiley & Sons, Inc.

Xiong, D., and Marble, D. 1996. Strategies for Real-Time Spatial Analysis Using Massively Parallel SIMD Computers: An Application to Urban Traffic Flow Analysis.

Zomaya, Albert Y. 1996. *Parallel and Distributed Computing Handbook*. New York: McGraw-Hill.

APPENDIX A

PROGRAM LISTING FOR GEODATA.H

```
/*
Copyright (c) 2000 Erik Shepard

Permission to use, copy, modify, distribute, and sell
this software and its documentation for any purpose is
hereby granted without fee, provided that (i) the above
copyright notices and this permission notice appear in
all copies of the software and related documentation,
and (ii) the name of Erik Shepard may not be used in
any advertising or publicity relating to the software
without the specific, prior written permission of
Erik Shepard.

THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY
OF ANY KIND, EXPRESS, IMPLIED OR OTHERWISE, INCLUDING
WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY
OR FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL ERIK SHEPARD BE LIABLE FOR ANY SPECIAL,
INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND,
OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER OR NOT ADVISED OF THE POSSIBILITY
OF DAMAGE, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF
OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
*/
/*
Copyright information for shapelib and libgeotiff:

 * Copyright (c) 1999, Frank Warmerdam
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL
```

```
 * THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR
 * OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
 * OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE
 * OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

 *    Written By: Niles D. Ritter.
 *
 *  copyright (c) 1995   Niles D. Ritter
 *
 *  Permission granted to use this software, so long as this copyright
 *  notice accompanies any products derived therefrom.

Copyright information for libtiff:

Copyright (c) 1988-1997 Sam Leffler
Copyright (c) 1991-1997 Silicon Graphics, Inc.

Permission to use, copy, modify, distribute, and sell this software and
its documentation for any purpose is hereby granted without fee,
provided that (i) the above copyright notices and this permission
notice appear in all copies of the software and related documentation,
and (ii) the names of Sam Leffler and Silicon Graphics may not be used
in any advertising or publicity relating to the software without the
specific, prior written permission of Sam Leffler and Silicon Graphics.

THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND,
EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY
WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL SAM LEFFLER OR SILICON GRAPHICS BE LIABLE FOR
ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND,
OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY THEORY
OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE
OF THIS SOFTWARE.
*/

/* geodata.h - Header files for data structures and data input
   and output. */

#include <stdio.h>
#include <sys/stat.h>
#include <geotiffio.h>
#include <xtiffio.h>
#include <shapefil.h>

/* Data structures */

typedef struct {
  long   count;
  double  xmin;
  double  ymin;
  double  xmax;
  double  ymax;
  double *data;
} Points;
```

```
typedef struct {
  long  rows;
  long  cols;
  double cellsize;
  double xmin;
  double ymin;
  double xmax;
  double ymax;
  unsigned short **data;
} Grid;

struct neighbor {
  long number;
  double distance;
  struct neighbor *next;
  struct neighbor *previous;
};

/* Macros for accessing points data */

#define X(i) data[3 * i]
#define Y(i) data[(3 * i) + 1]
#define Z(i) data[(3 * i) + 2]

/* Function prototypes */

/* Points and shapefiles; note that the analysis routines have direct
   access through fseek and fread to get points data.  There are not
   functions defined for random access to binary points files. */

int AllocatePoints(Points **pts, long count);
int FreePoints(Points **pts);
int LoadPoints(char *filename, Points **pts);
int SavePoints(Points *pts, char *filename);
int SaveCDF(Points *pts, char *filename);
int SaveShapefile(Points *pts, char shape[9]);

/* Grids */

int AllocateGrid(Grid **grid, long rows, long cols);
int FreeGrid(Grid **grid);
int LoadGrid(char *filename, Grid **grid);
int SaveGrid(Grid *grid, char *filename);int LoadTIFF(Grid **grid,
    char tiff[9]);
int SaveTIFF(Grid *grid, char tiff[9]);
```

APPENDIX B

PROGRAM LISTING FOR GEODATA.C

```
/*
Copyright (c) 2000 Erik Shepard

Permission to use, copy, modify, distribute, and sell
this software and its documentation for any purpose is
hereby granted without fee, provided that (i) the above
copyright notices and this permission notice appear in
all copies of the software and related documentation,
and (ii) the name of Erik Shepard may not be used in
any advertising or publicity relating to the software
without the specific, prior written permission of
Erik Shepard.

THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY
OF ANY KIND, EXPRESS, IMPLIED OR OTHERWISE, INCLUDING
WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY
OR FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL ERIK SHEPARD BE LIABLE FOR ANY SPECIAL,
INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND,
OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER OR NOT ADVISED OF THE POSSIBILITY
OF DAMAGE, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF
OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
*/

/* geodata.c - Implementations for data input and output. */

#include "geodata.h"

int AllocatePoints(Points **pts, long count) {

  Points *p;

  /* Allocate space for the Points structure and for the x, y and
     z arrays */

  if ((p = (Points *)malloc(sizeof (Points))) == NULL) return 1;
  p->count = count;
  if ((p->data = (double *)malloc(p->count * 3 * sizeof(double)))
     == NULL) return 1;
```

```
  /* Set the pointer */

  *pts = p;
  p = NULL;

  return 0;

}

int FreePoints(Points **pts) {

  Points *p;

  /* Set a reference to the points structure */

  p = *pts;

  /* Free the memory blocks of the structure */

  free(p->data);
  free(p);

  return 0;

}

int LoadPoints(char *filename, Points **pts) {

  FILE *fp;
  Points *p;
  long count;
  double bounds[4];

  /* Assign a local pointer */

  p = *pts;

  /* Name the file */

  if ((fp = fopen(filename, "r")) == NULL) {
    printf("\nCannot open points file %s.\n\n", filename);
    exit(1);
  }

  /* Read in the header and then the data */

  fseek(fp, 0, SEEK_SET);
  fread(&count, sizeof(long), 1, fp);
  fread(bounds, sizeof(double), 4, fp);
  AllocatePoints(&p, count);
  p->count = count;
  p->xmin = bounds[0];
  p->ymin = bounds[1];
  p->xmax = bounds[2];
  p->ymax = bounds[3];
  fread(p->data, sizeof(double), 3 * p->count, fp);
```

```
  /* Reset pointers */

  *pts = p;
  p = NULL;

  fclose(fp);

  return 0;

}

int SavePoints(Points *pts, char *filename) {

  FILE *fp;

  /* Save the points to a random access file to be used as our
     data source. */

  if ((fp = fopen(filename, "w")) == NULL) {
    printf("\nCannot open points file %s.\n\n", filename);
    exit(1);
  }

  /* Write header information. */

  fwrite(&(pts->count), sizeof(long), 1, fp);
  fwrite(&(pts->xmin), sizeof(double), 1, fp);
  fwrite(&(pts->ymin), sizeof(double), 1, fp);
  fwrite(&(pts->xmax), sizeof(double), 1, fp);
  fwrite(&(pts->ymax), sizeof(double), 1, fp);

  /* Write data values. */

  fwrite(pts->data, sizeof(double), 3 * pts->count, fp);

  fclose(fp);

  return 0;

}

int SaveCDF(Points *pts, char *filename) {

  FILE *fp;
  int i;

  /* Write to a comma delimited file. */

  if ((fp = fopen(filename, "w")) == NULL) {
    printf("\nCannot open text file %s.\n\n", filename);
    exit(2);
  }
  fprintf(fp, "count=%ld;\n", pts->count);
  fprintf(fp, "xmin=%f;\n", pts->xmin);
  fprintf(fp, "ymin=%f;\n", pts->ymin);
  fprintf(fp, "xmax=%f;\n", pts->xmax);
  fprintf(fp, "ymax=%f;\n", pts->ymax);
```

```
  for (i = 0; i < pts->count; i++) {
    fprintf(fp, "%f, %f, %f\n", pts->X(i), pts->Y(i),
      (double) pts->Z(i));
  }

  fclose(fp);
  return 0;

}

int SaveShapefile(Points *pts, char shape[9]) {

  long i;
  double x, y, z;
  char name[13];
  SHPObject *psShape;
  SHPHandle hSHP;
  DBFHandle hDBF;

  /* Name the output shapefile. */

  strcpy(name, shape);

  /* Create the point layer and associated DBF table. */

  hSHP = SHPCreate(name, SHPT_POINT);
  if (hSHP == NULL) {
    printf("\nCannot open %s.shp.\n\n", name);
    return 1;
  }
  hDBF = DBFCreate(strcat(name, ".dbf"));
  if (hDBF == NULL) {
    printf("\nCannot open %s.\n\n", name);
    return 1;
  }
  if (DBFAddField(hDBF, "Z", FTDouble, 18, 6) == -1) {
    printf("\nCannot add Z field to %s.\n\n", name);
    return 1;
  }
  if (DBFAddField(hDBF, "r", FTDouble, 18, 6) == -1) {
    printf("\nCannot add r field to %s.\n\n", name);
    return 1;
  }

  /* Loop through the data and write it to the shapefile.  */

  for (i = 0; i < pts->count; i++) {
    x = pts->X(i);
    y = pts->Y(i);
    z = (double) pts->Z(i);
    psShape = SHPCreateObject(SHPT_POINT, -1, 0, NULL, NULL, 1, &x, &y,
                              NULL, NULL);
    SHPWriteObject(hSHP, -1, psShape);
    SHPDestroyObject(psShape);
    DBFWriteDoubleAttribute(hDBF, i, 0, z);
    DBFWriteDoubleAttribute(hDBF, i, 1, i);
  }
```

```
  /* Close the shapefile and table. */

  SHPClose(hSHP);
  DBFClose(hDBF);

  return 0;

}

int AllocateGrid(Grid **grid, long rows, long cols) {

  Grid *g;
  long i;

  g = *grid;

  /* Allocate space for the input data grid. */

  if ((g = (Grid *)malloc(sizeof(Grid))) == NULL) return 1;

  /* Allocate space for the data block and read in and parse the
     data. */

  if ((g->data = (unsigned short **) malloc(rows *
     sizeof(unsigned short *))) == NULL) return 1;
  if ((g->data[0] = (unsigned short *) malloc(rows * cols *
     sizeof(unsigned short))) == NULL) return 1;
  for (i = 0; i < rows; i++) g->data[i] = g->data[0] + i * cols;

  /* Set the pointer */

  *grid = g;
  g = NULL;

  return 0;

}

int FreeGrid(Grid **grid) {

  Grid *g;

  /* Set a reference to the Grid structure */

  g = *grid;

  /* Free the pointers in and to the grid structure */

  free(g->data);
  free(g);

  return 0;

}

int LoadGrid(char *filename, Grid **grid) {
```

```
  FILE *fp;
  Grid *g;
  long rows, cols;
  double bounds[4], cellsize;
  int i;

  /* Assign a local pointer */

  g = *grid;

  /* Name the file */

  if ((fp = fopen(filename, "r")) == NULL) {
    printf("\nCannot open grid file %s.\n\n", filename);
    exit(1);
  }

  /* Read in the header and then the data */

  fseek(fp, 0, SEEK_SET);
  fread(&rows, sizeof(long), 1, fp);
  fread(&cols, sizeof(long), 1, fp);
  fread(&cellsize, sizeof(double), 1, fp);
  fread(bounds, sizeof(double), 4, fp);
  AllocateGrid(&g, rows, cols);
  g->rows = rows;
  g->cols = cols;
  g->cellsize = cellsize;
  g->xmin = bounds[0];
  g->ymin = bounds[1];
  g->xmax = bounds[2];
  g->ymax = bounds[3];
  for (i = 0; i < g->rows; i++)
    fread(g->data[i], sizeof(unsigned short), g->cols, fp);

  /* Reset pointers */

  *grid = g;
  g = NULL;

  fclose(fp);

  return 0;

}

int SaveGrid(Grid *grid, char *filename) {

  FILE *fp;
  long i;

  /* Save the grid to a random access file. */

  if ((fp = fopen(filename, "w")) == NULL) {
    printf("\nCannot open grid file %s.\n\n", filename);
    exit(1);
  }
```

```
  /* Write header information. */

  fwrite(&(grid->rows), sizeof(long), 1, fp);
  fwrite(&(grid->cols), sizeof(long), 1, fp);
  fwrite(&(grid->cellsize), sizeof(double), 1, fp);
  fwrite(&(grid->xmin), sizeof(double), 1, fp);
  fwrite(&(grid->ymin), sizeof(double), 1, fp);
  fwrite(&(grid->xmax), sizeof(double), 1, fp);
  fwrite(&(grid->ymax), sizeof(double), 1, fp);

  /* Write data values. */

  for (i = 0; i < grid->rows; i++)
    fwrite(grid->data[i], sizeof(unsigned short), grid->cols, fp);

  fclose(fp);

  return 0;

}

int LoadTIFF(Grid **grid, char tiff[9]) {

  char name[256];
  TIFF *tif=(TIFF*)0;
  GTIF *gtif=(GTIF*)0;
  long rows, cols, i;
  double *padfTiePoints, *padfScale;
  double bounds[4];
  int count;
  uint16 bps, spp;
  u_char *buf;
  Grid *g;

  /* Name the input tif file. */

  strcpy(name, getenv("GPDATA"));
  strcat(name, "/");
  strcat(name, tiff);
  strcat(name, ".tif");

  /* Load the TIFF headers and compute the boundaries */

  tif = XTIFFOpen(name, "r");
  if (!tif) {
    printf("Cannot open %s.\n", name);
    return 1;
  }
  gtif = GTIFNew(tif);
  if (!gtif) {
    printf("Cannot open %s.\n", name);
    return 1;
  }

  TIFFGetField(tif, TIFFTAG_IMAGEWIDTH, &cols);
  TIFFGetField(tif, TIFFTAG_IMAGELENGTH, &rows);
  TIFFGetField(tif, TIFFTAG_BITSPERSAMPLE, &bps);
```

```
TIFFGetField(tif, TIFFTAG_SAMPLESPERPIXEL, &spp);
TIFFGetField(tif, TIFFTAG_GEOPIXELSCALE, &count, &padfScale);
TIFFGetField(tif, TIFFTAG_GEOTIEPOINTS, &count, &padfTiePoints);
bounds[0] = padfTiePoints[3] - (padfScale[0] / 2);
bounds[3] = padfTiePoints[4] + (padfScale[1] / 2);
bounds[1] = bounds[3] - (padfScale[1] * rows);
bounds[2] = bounds[0] + (padfScale[0] * cols);

/* Make sure that the image is 16 bit (1 sample per pixel).
   Also make sure the pixel size is square. */

if (bps != 16 || spp != 1) {
 printf("\nInput image must be 16 bit with 1 sample per pixel.\n\n");
 return 1;
}

if (padfScale[0] != padfScale[1]) {
  printf("\nInput image pixel size is not square.\n\n");
  return 1;
}

/* Allocate space for the input data grid. */

AllocateGrid(&g, rows, cols);

/* Initialize the input grid. */

g->rows = rows;
g->cols = cols;
g->cellsize = padfScale[0];
g->xmin = bounds[0];
g->ymin = bounds[1];
g->xmax = bounds[2];
g->ymax = bounds[3];

/* Read in and parse the data. */

buf = (u_char *)_TIFFmalloc(TIFFScanlineSize(tif));
for (i = 0; i < rows; i++) {
  if (TIFFReadScanline(tif, buf, i, 0) < 0) {
    TIFFError("ReadImage", "Failure in ReadScanline\n");
    return 0;
  }
  memmove(g->data[i], buf, TIFFScanlineSize(tif));
}

/* Close the TIFF file and clean up memory */

GTIFFree(gtif);
XTIFFClose(tif);
_TIFFfree(buf);
```

```
  /* Set a reference to return the loaded grid */

  *grid = g;
  g = NULL;

  return 0;

}

int SaveTIFF(Grid *grid, char tiff[9]) {

  char name[13];
  TIFF *tif=(TIFF*)0;
  GTIF *gtif=(GTIF*)0;
  double tiepoints[6]={0, 0, 0, grid->xmin + (grid->cellsize / 2),
                       grid->ymax - (grid->cellsize / 2), 0.0};
  double pixscale[3]={grid->cellsize,grid->cellsize,0};
  long i;

  /* Name the output TIFF. */

  strcpy(name, tiff);
  strcat(name, ".tif");

  /* Set up the TIFF and GeoTIFF headers */

  tif = XTIFFOpen(name, "w");
  if (!tif) {
    printf("Cannot open %s.\n", name);
    return 1;
  }

  gtif = GTIFNew(tif);
  if (!gtif) {
    printf("Failure in GTIFNew.\n");
    TIFFClose(tif);
    return 2;
  }

  TIFFSetField(tif, TIFFTAG_IMAGEWIDTH, grid->cols);
  TIFFSetField(tif, TIFFTAG_IMAGELENGTH, grid->rows);
  TIFFSetField(tif, TIFFTAG_COMPRESSION, COMPRESSION_NONE);
  TIFFSetField(tif, TIFFTAG_PHOTOMETRIC, PHOTOMETRIC_MINISBLACK);
  TIFFSetField(tif, TIFFTAG_PLANARCONFIG, PLANARCONFIG_SEPARATE);
  TIFFSetField(tif, TIFFTAG_BITSPERSAMPLE, 16);
  TIFFSetField(tif, TIFFTAG_SAMPLESPERPIXEL, 1);
  TIFFSetField(tif, TIFFTAG_GEOTIEPOINTS, 6, tiepoints);
  TIFFSetField(tif, TIFFTAG_GEOPIXELSCALE, 3, pixscale);

  /* Write the scan lines one at a time */

  for (i = 0; i < grid->rows; i++) {
    if (!TIFFWriteScanline(tif, grid->data[i], i, 0)) {
      TIFFError("WriteImage", "Failure in WriteScanline\n");
      return 0;
    }
  }
```

```
  /* Close the TIFF file */

  GTIFWriteKeys(gtif);
  GTIFFree(gtif);
  XTIFFClose(tif);

  return 0;

}
```

APPENDIX C

PROGRAM LISTING FOR CREATEPOINTS.C

```
/*
Copyright (c) 2000 Erik Shepard

Permission to use, copy, modify, distribute, and sell
this software and its documentation for any purpose is
hereby granted without fee, provided that (i) the above
copyright notices and this permission notice appear in
all copies of the software and related documentation,
and (ii) the name of Erik Shepard may not be used in
any advertising or publicity relating to the software
without the specific, prior written permission of
Erik Shepard.

THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY
OF ANY KIND, EXPRESS, IMPLIED OR OTHERWISE, INCLUDING
WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY
OR FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL ERIK SHEPARD BE LIABLE FOR ANY SPECIAL,
INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND,
OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER OR NOT ADVISED OF THE POSSIBILITY
OF DAMAGE, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF
OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
*/


/* createpoints.c - Creates a random points file. */

#include <stdio.h>
#include <time.h>
#include <sys/stat.h>
#include <string.h>
#include <math.h>
#include "geodata.h"

int main(int argc, char *argv[]) {

  char filename[13];
  long i;
  char tiff[13];
  Points *points;
  Grid *grid = NULL;
```

```
long r, c;
float ro, co;
double v1, v2, yt;
double x, y, z;
double randm;
long count;

/* Check the arguments to make sure that an image and number of
   runs was specified.  If not, return an error to the user. */

if (argc < 4) {
  printf("\nUsage: createpoints <image> <number of points>
    <points>\n\n");
  exit(0);
}
strcpy(tiff, argv[1]);
count = atol(argv[2]);
strcpy(filename, argv[3]);

printf("Generating %s from %s with %ld points.\n", filename,
  tiff, count);

/* Load the input grid into memory */

LoadTIFF(&grid, tiff);

/* Allocate space for the points */

AllocatePoints(&points, count);

/* Generate random x and y coordinates independently to insure
   correct spectral properties (this requires 2 loops).  Use
   timeoday (in seconds) to seed the random number generator. */

randm = (double) grid->cols / (double)RAND_MAX;
for (i = 0; i < points->count; i++) {
  x = (rand() * randm);
  points->X(i) = x;
  if (i == 0) points->xmin = points->xmax = points->X(i);
  else {
    if (points->X(i) < points->xmin) points->xmin = points->X(i);
    if (points->X(i) > points->xmax) points->xmax = points->X(i);
  }
}
randm = (double) grid->rows / (double)RAND_MAX;
for (i = 0; i < points->count; i++) {
  y = (rand() * randm);
  points->Y(i) = y;
  if (i == 0) points->ymin = points->ymax = points->Y(i);
  else {
    if (points->Y(i) < points->ymin) points->ymin = points->Y(i);
    if (points->Y(i) > points->ymax) points->ymax = points->Y(i);
  }
}

/* Convert the boundary values from grid space to ground coordinates.
   Because y increases from bottom to top, we must invert ymin and
```

```
   Ymax to get the correctly mapped min and max y values. */

points->xmin = ((points->xmin * (grid->xmax - grid->xmin))
               / grid->cols) + grid->xmin;
points->xmax = ((points->xmax * (grid->xmax - grid->xmin))
               / grid->cols) + grid->xmin;
yt = grid->ymax - ((points->ymax * (grid->ymax - grid->ymin))
               / grid->rows);
points->ymax = grid->ymax - ((points->ymin
               * (grid->ymax - grid->ymin)) / grid->rows);
points->ymin = yt;

/* Interpolate values for the corresponding x and y positions. */

for (i = 0; i < points->count; i++) {

  /* Retrieve the random position. */

  x = points->X(i);
  y = points->Y(i);
  z = 0.0;

  /* Parse the coordinates into an integer and a floating point part.
     The integer will locate the base row and column.  The floating
     point will determine the neighboring rows and columns.  Offset
     by 0.5 to account for grid value at "center" of pixel. */

  co = (float) (x - 0.5) - (c = (x - 0.5));
  ro = (float) (y - 0.5) - (r = (y - 0.5));

  /* If the point is outside of the first or last rows or columns,
     handle in special case (linear interpolation if not on grid,
     or sample if on grid or in corner) */

  if ((r == 0 && ro <= 0) || (c == 0 && co <= 0) ||
       r == (grid->rows - 1) || c == (grid->cols - 1)) {
    if ((ro <= 0 && co <= 0) || (ro <= 0 && c == (grid->cols - 1))
         || (co <= 0 && r == (grid->rows - 1))
         || (c == (grid->cols - 1) && r == (grid->rows - 1))
         || (ro == 0) || (co == 0)) {
      z = grid->data[abs(r)][abs(c)];
    }
    else if (ro <= 0 || r == (grid->rows - 1)) {
      z = grid->data[abs(r)][abs(c)] + co * (grid->data[abs(r)]
            [abs(c + 1)] - grid->data[abs(r)][abs(c)]);
    }
    else if (co <= 0 || c == (grid->cols - 1)) {
        z = grid->data[abs(r)][abs(c)] + ro *
            (grid->data[abs(r + 1)][abs(c)]
            - grid->data[abs(r)][abs(c)]);
    }
  }

  /* If the point is on the grid, use a linear interpolation or a
     sample if the point falls on the lattice.  Otherwise, use a
     bilinear interpolation. */
```

```
    else {
      if (ro == 0 && co != 0)
        z = grid->data[r][c] + co * (grid->data[r][c + 1]
              - grid->data[r][c]);
      else if (ro != 0 && co == 0)
        z = grid->data[r][c] + ro * (grid->data[r + 1][c]
              - grid->data[r][c]);
      else if (ro == 0 && co == 0)
        z = grid->data[r][c];
      else {
        v1 = grid->data[r][c] + co * (grid->data[r][c + 1]
             - grid->data[r][c]);
        v2 = grid->data[r + 1][c] + co * (grid->data[r + 1][c + 1]
             - grid->data[r + 1][c]);
        z = v1 + ro * (v2 - v1);
      }
    }

    /* Assign the computed value to the point */

    points->X(i) =
      ((x * (grid->xmax - grid->xmin)) / grid->cols) + grid->xmin;
    points->Y(i) =
      grid->ymax - ((y * (grid->ymax - grid->ymin)) / grid->rows);
    points->Z(i) = z;

  }

  /* Write the points in a random access binary file and free
     memory. */

  SavePoints(points, filename);
  FreePoints(&points);

  /* Free dynamically allocated memory */

  FreeGrid(&grid);

  printf("Done!\n");

  return 0;

}
```

APPENDIX D

PROGRAM LISTING FOR CONVERT.C

```
/*
Copyright (c) 2000 Erik Shepard

Permission to use, copy, modify, distribute, and sell
this software and its documentation for any purpose is
hereby granted without fee, provided that (i) the above
copyright notices and this permission notice appear in
all copies of the software and related documentation,
and (ii) the name of Erik Shepard may not be used in
any advertising or publicity relating to the software
without the specific, prior written permission of
Erik Shepard.

THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY
OF ANY KIND, EXPRESS, IMPLIED OR OTHERWISE, INCLUDING
WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY
OR FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL ERIK SHEPARD BE LIABLE FOR ANY SPECIAL,
INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND,
OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER OR NOT ADVISED OF THE POSSIBILITY
OF DAMAGE, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF
OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
*/

/* convert.c - Converts binary points and grid files to shapefile, CDF,
   and GeoTIFF format respectively. */

#include <stdio.h>
#include <string.h>
#include <math.h>
#include "geodata.h"

int main(int argc, char *argv[]) {

  Points *points;
  Grid *grid;
  int run;
  char output[8], filename[13], format[6];

  /* Check the arguments for input and output names.  */

  if (argc < 4) {
```

```
    printf("\nUsage: convert <geodata> <shape | cdf | tiff>
      <filename>\n\n");
    exit(0);
  }
  run = atoi(argv[1]);
  strcpy(filename, argv[1]);
  strcpy(format, argv[2]);
  strcpy(output, argv[3]);

  /* Load the input and save to the correct format. */

  if (strcmp(format, "shape") == 0) {
    LoadPoints(filename, &points);
    SaveShapefile(points, output);
    FreePoints(&points);
  }
  else if (strcmp(format, "cdf") == 0) {
    LoadPoints(filename, &points);
    SaveCDF(points, output);
    FreePoints(&points);
  }
  else if (strcmp(format, "tiff") == 0) {
    LoadGrid(filename, &grid);
    SaveTIFF(grid, output);
    FreeGrid(&grid);
  }
  else {
    printf("%s is not a valid format.\n", format);
    exit(0);
  }

  return 0;

}
```

APPENDIX E

PROGRAM LISTING FOR PARALLEL.C

```
/*
Copyright (c) 2000 Erik Shepard

Permission to use, copy, modify, distribute, and sell
this software and its documentation for any purpose is
hereby granted without fee, provided that (i) the above
copyright notices and this permission notice appear in
all copies of the software and related documentation,
and (ii) the name of Erik Shepard may not be used in
any advertising or publicity relating to the software
without the specific, prior written permission of
Erik Shepard.

THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY
OF ANY KIND, EXPRESS, IMPLIED OR OTHERWISE, INCLUDING
WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY
OR FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL ERIK SHEPARD BE LIABLE FOR ANY SPECIAL,
INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND,
OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER OR NOT ADVISED OF THE POSSIBILITY
OF DAMAGE, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF
OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
*/

/* parallel.c - Perform parallel inverse distance weighting using
   Friedman's algorithm to partition data among the processors.
   We've avoided the overhead of the geodata structures by implementing
   direct read and write access. */

#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <sys/stat.h>
#include <sys/times.h>
#include <mpi.h>

/* Macros for accessing points data. */

#define X_(i) (3 * (i))
#define Y_(i) ((3 * (i)) + 1)
```

```
#define Z_(i) ((3 * (i)) + 2)
#define X(i) (3 * ((i) - first))
#define Y(i) ((3 * ((i) - first)) + 1)
#define Z(i) ((3 * ((i) - first)) + 2)

/* Structure for holding nearest neighbors. */

struct neighbor {
  long number;
  double distance;
  struct neighbor *next;
  struct neighbor *previous;
};

int main(int argc, char *argv[]) {

  int rank, procs, cellsize;
  MPI_Status status;
  long col, row, count, rows, cols, rf[3];
  double x, y, z, bounds[4];
  int nncount;
  char pointsfile[13];
  char gridfile[13];
  FILE *POINT_FP, *GRID_FP;
  unsigned short *raster;

  /* Initialize the MPI environment. */

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);

  /* Check the arguments and return a message from process 0, if
     necessary. */

  if (argc < 6) {
    if (rank == 0) {
      printf("\nUsage: parallel <points> <resolution> <grid>
              <neighbors> ");
      printf("<processors>\n\n");
    }
    MPI_Finalize();
    exit(0);
  }
  strcpy(pointsfile, argv[1]);
  cellsize = atoi(argv[2]);
  strcpy(gridfile, argv[3]);
  nncount = atoi(argv[4]);
  procs = atoi(argv[5]);

  /* Open the points input file and read the number of points, then
     compute grid properties.  On the root process, close the file - on
     the remaining processors, leave it open to read prototypes. */

  if ((POINT_FP = fopen(pointsfile, "r")) == NULL) {
    printf("\nCannot open file with sampled elevation values.\n\n");
    MPI_Finalize();
```

```
  exit(1);
 }
fseek(POINT_FP, 0, SEEK_SET);
fread(&count, sizeof(long), 1, POINT_FP);
fread(bounds, sizeof(double), 4, POINT_FP);
if (rank == 0) fclose(POINT_FP);
rows = (long)(((bounds[3] - bounds[1] +
       (cellsize / 2)) / cellsize) + 1);
cols = (long)(((bounds[2] - bounds[0] +
       (cellsize / 2)) / cellsize) + 1);
bounds[0] = bounds[0] - (cellsize / 2);
bounds[1] = bounds[1] - (cellsize / 2);
bounds[2] = bounds[0] + (cols * cellsize);
bounds[3] = bounds[1] + (rows * cellsize);

if (rank == 0) {

  /* The root process handles only creating and populating the grid
     file and starting/stopping the timer and writing to the
     logfile. */

  time_t start, finish;
  long runtime, i;
  FILE *LOG_FP;
  int proc;
  long rlength, rlengths[3], offset[3];
  double cellsize_d;

  /* Start the timer */

  time(&start);

  /* Initialize the grid file. */

  cellsize_d = (double) cellsize;
  if ((raster = (unsigned short *)
    malloc (cols * sizeof(unsigned short))) == NULL) {
      MPI_Finalize();
      exit(2);
  }
  if ((GRID_FP = fopen(gridfile, "w")) == NULL) {
    printf("\nCannot open grid file.\n\n");
    MPI_Finalize();
    exit(3);
  }
  fwrite(&rows, sizeof(long), 1, GRID_FP);
  fwrite(&cols, sizeof(long), 1, GRID_FP);
  fwrite(&cellsize_d, sizeof(double), 1, GRID_FP);
  fwrite(bounds, sizeof(double), 4, GRID_FP);
  for (row = 0; row < rows; row++)
    fwrite(raster, sizeof(unsigned short), cols, GRID_FP);

  for (proc = 1; proc < procs; proc++) {
    MPI_Recv(&rlength, 1, MPI_LONG, proc, 0, MPI_COMM_WORLD,
             &status);
    rlengths[proc - 1] = rlength;
    switch (proc) {
```

```
        case 1: offset[0] = 0; break;
        case 2: offset[1] = rlengths[0]; break;
        case 3: offset[2] = rlengths[0] + rlengths[1]; break;
      }
    }

    /* Collect partial rasters as they come in and write them to the
       grid. A first message will arrive with [proc, row, start_col,
       count].  We'll read a second message from processor proc with
       length count] and will write a raster for row row starting at
       column start_col]. */

    for (i = 0; i < ((procs - 1) * rows); i++) {
      MPI_Recv(rf, 3, MPI_LONG, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
        &status);
      MPI_Recv(raster, rlengths[rf[0] - 1], MPI_SHORT, rf[0], 0,
        MPI_COMM_WORLD, &status);
      fseek(GRID_FP, ((2 * sizeof(long)) + (5 * sizeof(double))
        + ((rf[1] * cols) + offset[rf[0] - 1]) *
        sizeof(unsigned short)),
        SEEK_SET);
      fwrite(raster, sizeof(unsigned short), rlengths[rf[0] - 1],
        GRID_FP);
      if ((rf[1] % 100) == 0)
        printf("Received row %ld from processor %ld\n", rf[1], rf[0]);
    }

    /* Close the grid file. */

    free(raster);
    fclose(GRID_FP);

    /* Stop the timer and write elapsed seconds to the log. */

    time(&finish);
    runtime = (long) difftime(finish, start);
    if ((LOG_FP = fopen("logfile", "a")) == NULL) {
      printf("\nCannot open log file for writing.\n\n");
      MPI_Finalize();
      exit(6);
    }
    fprintf(LOG_FP, "%ld, %ld, %ld\n", count, (rows * cols), runtime);
    fclose(LOG_FP);

  }

  else {

    /* The remaining processors handle loading and sorting the data and
       computing the nearest neighbors. */

    long divis, max_points, first, last, scount, i, j, incr;
    long dp, dfp, dsp, start, end, low, high, pos;
    double *data_pass, *data_merge, *data_below = NULL;
    double *data_above, *data, scond, *data_hold = NULL;
    long *seeds, rlength;
    struct neighbor *n_head, *n_tail, *n_curr, *n_shift;
```

```
double psqr, d1, p1;
double num, denom, weight;

/* Do some simple error checking and load the points. */

divis = ceil((float) count / (float) (procs - 1));
max_points = ceil((sqrt(32) / sqrt(3.14159265358979)) *
  sqrt(2 * count));
if (max_points > divis) {
  exit(11);
  MPI_Finalize();
}
if ((data = (double *)malloc(3 * divis * sizeof(double))) == NULL)
{
   MPI_Finalize();
   exit(7);
}
if (rank != (procs - 1)) {
  first = ((rank - 1) * divis);
  last = ((rank * divis) - 1);
  scount = divis;
}
else {
  first = ((rank - 1) * divis);
  last = count - 1;
  scount = (last - first) + 1;
}
fseek(POINT_FP, sizeof(long) + 4 * sizeof(double) +
               (3 * first * sizeof(double)), SEEK_SET);
fread(data, sizeof(double), 3 * scount, POINT_FP);
if (scount < divis)
  for (i = (3 * scount); i < (3 * divis); i++) data[i] = HUGE_VAL;

/* To begin the merge-splitting sort, first sort the local set. */

incr = divis / 2;
while (incr >= 1) {
  for (j = incr; j < divis; j++) {
    x = data[X_(j)];
    y = data[Y_(j)];
    z = data[Z_(j)];
    i = j;
    while (i >= incr && x < data[X_((i - incr))]) {
      data[X_(i)] = data[X_((i - incr))];
      data[Y_(i)] = data[Y_((i - incr))];
      data[Z_(i)] = data[Z_((i - incr))];
      i -= incr;
    }
    data[X_(i)] = x;
    data[Y_(i)] = y;
    data[Z_(i)] = z;
  }
  incr /= 2;
}

/* Begin the merge-split (parallel) procedure.  We will use a
   second array to do the merge and then split the array back in
```

```
   half. */

if (((data_pass = (double *)malloc(3 * divis *
   sizeof(double))) == NULL) ||
   ((data_merge = (double *)malloc(6 * divis *
   sizeof(double))) == NULL)) {
     MPI_Finalize();
     exit(9);
}
for (i = 1; i <= (int) ceil((float)(procs - 1) / (float)2); i++) {
  if ((rank % 2) == 1) {
    if (rank <= (2 * floor((procs - 1) / 2) - 1)) {
      MPI_Recv(data_pass, 3 * divis, MPI_DOUBLE, rank + 1, 0,
              MPI_COMM_WORLD, &status);
      dp = dfp = dsp = 0;
      while (dp != divis && dfp != divis) {
        if (data[X_(dp)] < data_pass[X_(dfp)]) {
          data_merge[X_(dsp)] = data[X_(dp)];
          data_merge[Y_(dsp)] = data[Y_(dp)];
          data_merge[Z_(dsp)] = data[Z_(dp)];
          dp++;
        }
        else {
          data_merge[X_(dsp)] = data_pass[X_(dfp)];
          data_merge[Y_(dsp)] = data_pass[Y_(dfp)];
          data_merge[Z_(dsp)] = data_pass[Z_(dfp)];
          dfp++;
        }
        dsp++;
      }
      if (dp < divis)
        memcpy(&data_merge[X_(dsp)], &data[X_(dp)],
          3 * (divis - dp) * sizeof(double));
      else
        memcpy(&data_merge[X_(dsp)], &data_pass[X_(dfp)],
          3 * (divis - dfp) * sizeof(double));
      memcpy(&data[X_(0)], &data_merge[X_(0)],
          3 * divis * sizeof(double));
      memcpy(&data_pass[X_(0)], &data_merge[X_(divis)],
          3 * divis * sizeof(double));
      MPI_Send(data_pass, 3 * divis, MPI_DOUBLE, rank + 1, 0,
              MPI_COMM_WORLD);
    }
    if (rank != 1) {
      MPI_Sendrecv(data, 3 * divis, MPI_DOUBLE, rank - 1, 0,
                   data, 3 * divis, MPI_DOUBLE, rank - 1, 0,
                   MPI_COMM_WORLD, &status);
    }
  }
  else {
    MPI_Sendrecv(data, 3 * divis, MPI_DOUBLE, rank - 1, 0,
                 data, 3 * divis, MPI_DOUBLE, rank - 1, 0,
                 MPI_COMM_WORLD, &status);
    if (rank <= (2 * floor((procs - 2) / 2))) {
      MPI_Recv(data_pass, 3 * divis, MPI_DOUBLE, rank + 1, 0,
              MPI_COMM_WORLD, &status);
      dp = dfp = dsp = 0;
```

```
      while (dp != divis && dfp != divis) {
        if (data[X_(dp)] < data_pass[X_(dfp)]) {
          data_merge[X_(dsp)] = data[X_(dp)];
          data_merge[Y_(dsp)] = data[Y_(dp)];
          data_merge[Z_(dsp)] = data[Z_(dp)];
          dp++;
        }
        else {
          data_merge[X_(dsp)] = data_pass[X_(dfp)];
          data_merge[Y_(dsp)] = data_pass[Y_(dfp)];
          data_merge[Z_(dsp)] = data_pass[Z_(dfp)];
          dfp++;
        }
        dsp++;
      }
      if (dp < divis)
        memcpy(&data_merge[X_(dsp)], &data[X_(dp)],
          3 * (divis - dp) * sizeof(double));
      else
        memcpy(&data_merge[X_(dsp)], &data_pass[X_(dfp)],
          3 * (divis - dfp) * sizeof(double));
      memcpy(&data[X_(0)], &data_merge[X_(0)],
          3 * divis * sizeof(double));
      memcpy(&data_pass[X_(0)], &data_merge[X_(divis)],
          3 * divis * sizeof(double));
      MPI_Send(data_pass, 3 * divis, MPI_DOUBLE, rank + 1, 0,
              MPI_COMM_WORLD);
    }
  }
}
free(data_merge);
free(data_pass);

/* Send the additional data necessary to pad the data array */

if (((data_above = (double *)malloc(
  3 * max_points * sizeof(double)))
  == NULL) || ((data_below = (double *)malloc(3 * max_points *
 sizeof(double))) == NULL)) {
   MPI_Finalize();
   exit(10);
}
if (rank > 1) {
  memcpy(data_above, data, 3 * max_points * sizeof(double));
  MPI_Send(data_above, 3 * max_points, MPI_DOUBLE, rank - 1, 0,
          MPI_COMM_WORLD);
}
if (rank < (procs - 1)) {
  memcpy(data_below, &data[X_(divis - max_points)],
          3 * max_points * sizeof(double));
  MPI_Send(data_below, 3 * max_points, MPI_DOUBLE, rank + 1, 0,
          MPI_COMM_WORLD);
}
if (rank > 1)
  MPI_Recv(data_below, 3 * max_points, MPI_DOUBLE, rank - 1, 0,
          MPI_COMM_WORLD, &status);
else
```

```
  free(data_below);
if (rank < (procs - 1))
  MPI_Recv(data_above, 3 * max_points, MPI_DOUBLE, rank + 1, 0,
          MPI_COMM_WORLD, &status);
else
  free(data_above);

/* Do a binary search to determine the first seed below the last
   point of the data_below array.  Start searching for seeds from
   this point.  We will store prototype seeds in a second array of
   length raster; although a fair amount of space will be wasted,
   it means that we can index seeds directly by column number,
   without manipulation. */

if ((seeds = (long *) malloc (cols * sizeof(long))) == NULL) {
    MPI_Finalize();
    exit(2);
}
start = end = 0;
if (rank > 1) {
  low = 0;
  high = (cols - 1);
  do {
    pos = floor((low + high) / 2);
    if (data_below[X_(max_points - 1)] >
      (bounds[0] + ((pos + 0.5) * cellsize))) low = pos;
    else high = pos;
  } while ((high - low) != 1);
  start = ((bounds[0] + ((low + 0.5) * cellsize)) >
     data_below[X_(max_points - 1)]) ? low : high;
}
else start = 0;
if (rank < (procs - 1))
  scond = data_above[X_(0)];
else
  scond = data[X_(scount - 1)];
col = start;
while ((x = bounds[0] + ((col + 0.5) * cellsize)) < scond) {
  if ((rank > 1) && (fabs(x - data_below[X_(max_points - 1)]) <
      fabs(x - data[X_(0)]))) { start++; continue; }
  else if ((rank < (procs - 1)) && (fabs(x - data_above[X_(0)]) <
      fabs(x - data[X_(scount - 1)]))) continue;
  low = 0;
  high = scount - 1;
  do {
    pos = floor((low + high) / 2);
    if (x > data[X_(pos)]) low = pos;
    else high = pos;
  } while ((high - low) != 1);
  seeds[col] = (fabs(x - data[X_(low)]) <
    fabs(x - data[X_(high)])) ? low + first : high + first;
  end = col++;
}

/* Concatenate the data, data_above and data_below arrays. */

data_hold = data;
```

```
if (rank == 1) {
  if ((data = (double *)malloc((3 * (scount + max_points)) *
    sizeof(double))) == NULL) {
      MPI_Finalize();
      exit(7);
  }
  memcpy(data, data_hold, 3 * scount * sizeof(double));
  memcpy(&data[scount], data_above,
    3 * max_points * sizeof(double));
  last = last + max_points;
}
else if (rank == (procs - 1)) {
  if ((data = (double *)malloc((3 * (scount + max_points)) *
    sizeof(double))) == NULL) {
      MPI_Finalize();
      exit(7);
  }
  memcpy(data, data_below, 3 * max_points * sizeof(double));
  memcpy(&data[max_points], data_hold,
    3 * scount * sizeof(double));
  first = first - max_points;
}
else {
  if ((data = (double *)malloc((3 * (scount + (2 * max_points))) *
    sizeof(double))) == NULL) {
      MPI_Finalize();
      exit(7);
  }
  memcpy(data, data_below, 3 * max_points * sizeof(double));
  memcpy(&data[max_points], data_hold,
    3 * scount * sizeof(double));
  memcpy(&data[3 * (max_points + scount)], data_above,
    3 * max_points *
     sizeof(double));
  first = first - max_points;
  last = last + max_points;
}
data_hold = NULL;
free(data_above);
free(data_below);

/* Allocate a doubly linked list with nncount nodes for the
   neighbors. */

if ((n_tail = n_head = (struct neighbor *) malloc
    (sizeof (struct neighbor))) == NULL) {
       MPI_Finalize();
       exit(0);
}
n_head->next = n_head->previous = NULL;
for (i = 1; i < nncount; i++) {
  if ((n_curr = (struct neighbor *) malloc
     (sizeof (struct neighbor))) == NULL) {
        MPI_Finalize();
        exit(13);
  }
  n_curr->next = NULL;
```

```
  n_curr->previous = n_tail;
  n_tail = n_tail->next = n_curr;
}

/* Begin processing the rows and writing to a partial raster.
   Return the partial raster at the end of each row. */

rlength = (end - start) + 1;
MPI_Send(&rlength, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD);
rf[0] = rank;

/* Allocate a partial raster. */

if ((raster = (unsigned short *)
  malloc (rlength * sizeof(unsigned short))) == NULL) {
    MPI_Finalize();
    exit(14);
}
for (row = 0; row < rows; row++) {

  y = bounds[3] - ((row + 0.5) * cellsize);

  /* Proces each of the cells for which this processor has
     seeds. */

  for (col = start; col <= end; col++) {

    x = bounds[0] + ((col + 0.5) * cellsize);

    /* Initialize the partitioned neighbor list. */

    n_curr = n_head;
    while (n_curr != NULL) {
      n_curr->distance = -1;
      n_curr = n_curr->next;
    }

    /* Retrieve the seed point and add it to the neighbors list */

    n_head->number = pos = seeds[col];
    n_head->distance = pow((data[Y(pos)] - y), 2) +
                       (psqr = pow((data[X(pos)] - x), 2));
    high = (pos <= last) ? pos + 1 : pos;
    low = (pos >= first) ? pos - 1 : pos;

    /* Find the 12 nearest neighbors starting with the seed point
       and store their distances. */

    while (n_tail->distance > psqr || n_tail->distance == -1) {

      /* Check to see whether the high or low search point is
         closer in projected distance. */

      pos = (fabs(data[X(low)] - x)
        <= fabs(data[X(high)] - x)) ? low : high;

      /* Adjust the new search maximum and minimums. */
```

```
    if (low == (first - 1) && high == (last + 1)) break;
    else {
      if (low > (first - 1) && pos == low) low--;
      else {
        if (high < (last + 1) && pos == high) high++;
        else if (low > (first - 1)) low--;
      }
    }

    /* Insert the neighbor into the neighbor list, if
       appropriate, in order of distance^2. */

    if ((p1 = pow((data[X(pos)] - x), 2)) > psqr) psqr = p1;
    d1 = pow((data[Y(pos)] - y), 2) + p1;
    if (d1 < n_tail->distance || n_tail->distance == -1) {
      n_curr = n_head;
      while (n_curr != NULL) {
        if (d1 < n_curr->distance || n_curr->distance == -1) {
          n_tail->number = pos;
          n_tail->distance = d1;
          if (n_curr != n_tail) {
            n_shift = n_tail;
            n_tail = n_tail->previous;
            n_tail->next = n_shift->previous = NULL;
            if (n_curr == n_head) n_head = n_shift;
            else {
              n_shift->previous = n_curr->previous;
              n_shift->previous->next = n_shift;
            }
            n_shift->next = n_curr;
            n_curr->previous = n_shift;
          }
          break;
        }
        n_curr = n_curr->next;
      }
    }
  }

  /* Compute the interpolated value and store it in the
     raster. */

  n_curr = n_head;
  num = denom = weight = 0;
  while (n_curr != NULL) {
    weight = (n_curr->distance == 0) ? 1 : 1 / n_curr->distance;
    num += data[Z(n_curr->number)] * weight;
    denom += weight;
    n_curr = n_curr->next;
  }
  raster[col - start] = (short) (num / denom);
}

rf[1] = row;
MPI_Send(rf, 3, MPI_LONG, 0, 0, MPI_COMM_WORLD);
MPI_Send(raster, rlength, MPI_SHORT, 0, 0, MPI_COMM_WORLD);
```

```
    }

    /* Clean up */

    free(data);
    free(raster);
    free(seeds);

  }

  /* Shutdown the MPI environment. */

  MPI_Finalize();

  return 0;

}
```