Inverse-Distance-Weighted Spatial Interpolation Using Parallel Supercomputers

Marc P. Armstrong and Richard Marciano*

Abstract

Interpolation is a computationally intensive activity that may require hours of execution time to produce results when large problems are considered. In this paper we describe a strategy to reduce computation times through the use of parallel processing. To acheive this goal, a serial algorithm that performs two-dimensional inverse-distance-weighted interpolation is decomposed into a form suitable for parallel processing in two shared memory computing environments. The first uses a conventional architecture with a single monolithic memory, while the second uses a hierarchically organized collection of local caches to implement a large shared virtual address space. A series of computational experiments was conducted in which the number of processors used in parallel is systematically increased. The results show a substantial reduction in total processing time and speedups that are close to linear when the additional processors are used. The general approach described in this paper can be used to improve the performance of other types of computationally intensive interpolation problems.

Introduction

The computation of a two-dimensional gridded surface from a dispersed set of control points with known values is a fundamental operation in automated cartography. Though several methods have been developed to accomplish this task (Lam, 1983; Burrough, 1986), inverse-distance-weighted interpolation is widely applied and is available in many commercial GIS software environments. For large problems, however, inverse-distance-weighted interpolation can require substantial amounts of computation. The purpose of this paper is to demonstrate how parallel processing can be used to improve the computational performance of an inverse-distance-weighted interpolation algorithm when it is applied to a large problem.

Background

The underlying assumption of inverse-distance-weighted interpolation is that of positive spatial autocorrelation (Cromley, 1992): The contribution of control points near to a grid location with an unknown value is greater than that of distant control points. This assumption is embedded in the following equation:

$$z_j = \frac{\sum_{i=1}^{N} w_{ij} z_i}{\sum_{i=1}^{N} w_{ij}}$$

where

z_j is the estimated value at grid location j,
z_i is the known value at control point location i, and
w_{ij} is the weight that controls the effect of control points on the calculation of z_j.

Often w_{ij} is set equal to d_{ij}^{-a}, where d_{ij} is some measure of distance and a is often 1.0 or 2.0 (Hodgson, 1992). As the value of the exponent (a) increases, close control points contribute a greater proportion to the value of each interpolated grid location (Mitchell, 1977, p. 256).

In this formulation, all control points (z_i) would contribute to the calculation of an interpolated value at cell z_j. Given the assumption of positive spatial autocorrelation, however, it is common to restrict computation to some neighborhood (k) of z_j. This is often done by setting a limit on the number of points used to compute the z_j values. The choice of an appropriate value for k has been the source of some discussion in the literature (Hodgson, 1992). The now-ancient SYMAP algorithm (Shepard, 1968), for example, attempts to ensure that between four and ten data points are used (Monmonier, 1982) by varying the search radius about each z_j. If fewer than four points are found within an initial radius, the radius is expanded; if too many (e.g., more than ten) points are found, the radius is contracted. MacDougall (1976) also implements a similar approach to neighborhood specification.

While this process is conceptually rational, it involves considerable computation because, for each grid point, the distance between it and all control points must be evaluated to determine those that are closest. MacDougall (1984), for example, demonstrated that computation times increased dramatically with the number of points used to interpolate a 24 by 80 map grid; this grid size was selected because it was the resolution of alphanumeric CRT displays popular at that time. Using an 8-bit, 2-MHz microcomputer and interpreted BASIC, he found that computation times increased from a little more than 30 minutes when three control points were used, to almost **13 hours** when calculations were based on 100 control points. Though most workstations and mainframes would now compute this small problem in a few seconds, the need for high performance computing can be

Departments of Geography and Computer Science and Program in Applied Mathematical and Computational Sciences; and Gerard Weeg Computer Center and Department of Computer Science, respectively; 316 Jessup Hall, The University of Iowa, Iowa City, IA 52242.
*R. Marciano is presently with the National Supercomputer Center for Energy and the Environment, University of Nevada, Las Vegas, Las Vegas, NV 89154.

Photogrammetric Engineering & Remote Sensing, Vol. 60, No. 9, September 1994, pp. 1097–1103.

0099-1112/94/0009-000$3.00/0

established by increasing the problem size from 24 by 80 to a larger 240 by 800 grid with an increased number of control points (10,000); this roughly maintains the same ratio of control points to grid points used by MacDougall (1984). When the problem is enlarged in this way, a similar computational barrier is encountered: using a fast workstation, an RS/6000-550, and an interpolation algorithm that uses the three closest control points, this larger and more realistic problem required 7 hours and 27 minutes execution time. This is consistent with many historical trends in computing; as machine speeds increase, so does the size of the problems we wish to solve, and, consequently, there is a continuing need to reduce computation times (Freund and Siegel, 1993).

The interpolation process described above has been called a "brute-force" approach (Hodgson, 1989), and researchers have worked to overcome the computational intractibilities associated with this method. White (1984) demonstrated the performance advantage of using integer (as opposed to floating point) distance calculations. MacDougall (1984) showed how computation times could be reduced by using a distance estimation technique that does not require the calculation of square roots. This estimation technique, however, introduces small errors into the interpolation process because distances along diagonals are underestimated by approximately 10 percent.

Other efforts have focused on reducing the number of computations made. Originally, the neighborhood for interpolation was established by computing distances between each grid point and all control points and then ordering these distances to find the *k*-nearest. According to Hodgson (1989), ERDAS adopted a strategy to reduce computation that uses a local approach to interpolation; a 3 by 3 window is located over each grid cell for which a value must be interpolated, and only those control points that are in the window are candidates for inclusion in the computation of the interpolated cell value. Hodgson (1989), however, states that, while this approach reduces computation, it is not guaranteed to find the *k*-nearest neighbors. Clarke (1990, p. 207) discusses this problem further and provides C code to implement this local approach to interpolation.

Hodgson (1989) developed an efficient interpolation strategy that is guaranteed to find the *k*-nearest points of a grid location using a "learned search" method. This method exploits the spatial structure inherent in the arrangement of control points; for any given cell, its *k*-nearest control points are likely to be the same as those of an adjoining cell. Therefore, rather than discard information about control points obtained during a previous search for neighbors from adjacent cells, his method retains these values, bases the new search on them, and reduces the total number of calculations required.

Despite these improvements, substantial amounts of computation time are still required for extremely large problems. Large interpolation problems are destined to become commonplace given the increasing diversity, size, and disaggregation of digital spatial databases. Parallel algorithms are often developed specifically to overcome the computational intractabilities that are associated with such large problems. It is interesting to note, however, that the efficient approach described by Hodgson (1989) was developed for implementation on a sequential (Von Neumann) machine; it does not appear to be well-suited for implememtation in parallel computing environments because it relies on the presence of serial dependencies in the data to reduce computation. New approaches must be developed to exploit the latent parallelism in spatial problems because it is likely that future high performance computing environments will be constructed using parallel architectures and methods of computation.

The parallel algorithms described in this paper are based on an existing serial brute force algorithm. Specifically, we demonstrate how a serial FORTRAN implementation of code that performs two-dimensional interpolation (MacDougall, 1984) is translated, using parallel programming extensions to FORTRAN 77, into versions that run in two parallel computing environments: a traditional shared-memory computer and a newer, more massively parallel hybrid architecture that uses a distributed hierarchical cache structure to implement shared-memory access. In translating a serial program into a form suitable for parallel processing, the characteristics of the problem and the architecture of the computer to be used must be considered. We first briefly discuss architectural factors and then turn to a specific discussion of our parallel implementations of the interpolation algorithm. Note that these "brute-force" implementations represent worst case scenarios against which other approaches to parallel algorithm enhancement can be compared.

Architectural Considerations

During the past several years, many computer architects and manufacturers have turned from pipelined architectures toward parallel processing as a means of providing cost-effective high performance computing environments (El-Rewini *et al.*, 1993; Myers, 1993; Pancake, 1991). Though parallel architectures take many forms, a basic distinction can be drawn on the basis of the number of instructions that are executed in parallel. A single instruction, multiple data (SIMD) stream computer simultaneously executes the same instruction on several (often thousands of) data items. Consequently, programs developed for such architectures normally use a synchronous, fine-grained approach to parallelism. A multiple instruction, multiple data (MIMD) stream computer, on the other hand, handles the partitioning of work among processors in a more flexible way, because processors can be allocated tasks that vary in size. For example, programmers might assign either portions of a large loop or a copy of an entire procedure to each processor. This asynchronous approach to parallelism is often implemented using a shared-memory model in which each processor can independently read from and write to a common address space. This general approach was adopted for the computational experiments reported in this paper.

The allocation and execution of asynchronous parallel processes can occur using either an architecture explicitly designed for parallel processing, such as the computers used in our experiments, or a loosely confederated set of networked workstations using software such as LINDA (Carriero and Gelernter, 1990) or PVM (Parallel Virtual Machine) (Beguelin *et al.*, 1991; Beguelin *et al.*, 1993). LINDA and PVM are coordination languages that enable parallel processing on these networked computers. Because tasks can be assigned with different amounts of computation when this coarse-grained MIMD approach is used, programmers must be concerned with balancing workloads across processors. If a given processor finishes with it assigned task and it requires data being computed by another processor in order to continue, then a dependency is said to exist, the pool of available processors is being used inefficiently, and parallel performance will be degraded (Armstrong and Densham, 1992).

Code Fragment 1. EPF-based pseudocode for parallel interpolation.

```
DECLARATIONS
FORMAT STATEMENTS
FILE MANAGEMENT
READ DATA POINTS
CALCULATE INTERPOLATION PARAMETERS
   t_start = etime(tmp)
   FOR EACH CELL IN THE MAP
   PARALLEL
     INTEGER I, J, K, rad
     REAL   TEMP, T, B
     REAL   distance(Maxcoords)
     INTEGER L(Maxcoords)
     PRIVATE I, J, K, RAD, TEMP, T, B, distance, L
     DOALL (J=1:Columns)
       DO 710 I=1, Rows
          FOR EACH DATA POINT
          COMPUTE DISTANCE FROM POINT TO GRID CELL
          CHECK NUMBER OF POINTS WITHIN RADIUS
          COMPUTE INTERPOLATED VALUE
710   CONTINUE
     END DOALL
   END PARALLEL
  t_end = etime(tmp)
  t1 = (t_end - t_start)
END
```

Implementation

The computational experiments described in this paper were performed using two parallel computers: an Encore Multimax and a KSR1 supercomputer. The Encore is a modular MIMD computing environment that uses a conventional shared memory architecture (Brawer, 1989, pp. 28-29). Users can access between 1 and 14 NS32332 processors that are connected by a Nanobus with a sustained bandwidth of 100 megabytes per second. Each processor has its own small (32-K byte) cache of fast static RAM and can access 32 Mbytes of fast, global, shared memory over the Nanobus.

Unlike the Encore, the KSR1 supercomputer does not have a single, monolithic shared memory. Instead, it uses a distributed architecture. This distributed architecture can be programmed like a conventional shared memory machine, but it is actually built from a collection of local caches. The basic configuration of processing units of the KSR1 that we used at the Cornell Theory Center consists of 64 separate RISC processors that are organized into two rings of 32 processors; each processor is connected to a high-bandwidth network. Memory access in this distributed structure, known as ALLCACHE, is organized hierarchically. Each processor in the architecture has a 256-KB subcache and a 32-MB cache. Very fast access is provided for local memory requests (e.g., from the subcache), and increasing latency penalties are incurred when memory must be accessed from remote locations. Specifically, data can be accessed from the subcache of each processor in only two clock cycles. If a process requires data that are not found in the subcache, they might be accessed from the 32-MB cache, and a penalty of 18 clock cycles would be incurred. However, if data must be accessed from an entirely different processor on the network, 175 clock cycles are required. Even though the machine can be programmed as a single large shared virtual space, because of these latency penalties, workload distribution and dependency relations must be carefully evaluated during the algorithm design stage.

Our parallel implementations of MacDougall's serial interpolation algorithm use parallel programming supersets of FORTRAN 77 that support parallel task creation and control, including memory access and task synchronization (e.g., Brawer, 1989). Code Fragments 1 and 2, based on MacDougall (1984), present two simplified (pseudocode) views of the brute-force interpolation algorithm. Code Fragment 1 was developed for the Encore environment and uses Encore Parallel FORTRAN (EPF) constructs to implement parallelism (Encore Computer, 1988). Code Fragment 2 was implemented for the KSR1 supercomputer and uses extensions that are similar in function to those provided by EPF. In general, the implementations vary primarily in syntax and are briefly described in the following sections.

Parallel Task Creation

Each parallel implementation is a conventional FORTRAN 77 program in which parallel constructs are inserted. In the Encore environment, EPF constructs can only be used within parallel regions which are defined by enclosing code blocks with the keywords **PARALLEL** and **END PARALLEL**. Within a parallel region, the program initiates additional tasks and executes them in parallel. For example, the **DOALL** ... **END DOALL** construct partitions loop iterations among the available processors (Code Fragment 1).

In the KSR1 programming environment, a **TILE** directive is used to control the partitioning of tasks among processors (Code Fragment 2). This directive automatically allocates tasks among the available set of processors using, in this case, the loop iterations controlled by the J counter, which are assigned to different processors that are used to compute independently each element of the shared interpolated matrix.

Shared Memory

In the EPF environment, all variables are shared among the set of parallel tasks unless they are explicitly re-declared inside a parallel region. A variable declared, or re-declared, inside a parallel region cannot be accessed outside that parallel region; therefore, each task has its own private copy of that

Code Fragment 2. KSR1 pseudocode for the interpolatin problem. The emphasis in this example is on the main loop in the interpolation code.

```
   integer L(Maxcoords)
   REAL  distance(Maxcoords)

   common /myblock/ distance, L
c*ksr* psc /myblock/

c*ksr* TILE (J, PRIVATE=(I,K,K1,K2,L1,rad,temp,T,B,/myblock/))
      DO 700 J=1, Columns
        DO 710 I=1, Rows
.....
710      CONTINUE
700   CONTINUE
```

Notes:

1) **psc** stands for partially shared common.

2) A **c*ksr* psc** declaration allows each parallel thread in a **TILE** directive to have its own copy of the common block. This is required because the **PRIVATE** declaration does not allow the use of multidimensional variables.

TABLE 1. COMPUTATION TIMES WHEN DIFFERENT NUMBERS OF ENCORE PROCESSORS ARE USED TO COMPUTE THE 240 BY 800 GRID USING 10,000 CONTROL POINTS AND THREE PROXIMAL POINTS.

Processors	1	2	4	6	8	10	12	14
Hours	33.3	17.0	8.5	5.7	4.2	3.4	2.8	2.5

variable. This behavior is explicitly enforced in both environments by using the **PRIVATE** keyword when declaring variables. This approach could be used, for example, in a case when each of several parallel tasks needs its own copy of data to modify specific elements in a shared data structure. In the KSR1 environment, however, the **PRIVATE** construct does not support multidimensional variables. Consequently, we used an extension, called partially shared common, that enabled us to implement the independent calculation of interpolated grid values (Code Fragment 2).

General Strategy for Implementation Using Shared Memory

The process of converting a serial program to a parallel version often takes place in a series of steps, beginning with an analysis of dependencies among program components; such dependencies may preclude efficient implementation (Armstrong and Densham, 1992). As the code is decomposed into discrete parts, each is treated as an independent process that can be executed concurrently on different processors. While some problems may need to be completely restructured to achieve an efficient parallel implementation, in many instances conversion is straightforward. In this case, the interpolation algorithm can be cleanly divided into a set of independent processes using a coarse-grained approach to parallelism in which the computation of an interpolated value for each grid cell in the lattice is treated independently from the computation of values for all other cells. Though some variables are declared private to the computation of each cell value, the matrix that contains the interpolated grid values is shared. Thus, each process contributes to the computation of the grid by independently writing its results to the matrix held in shared memory. In principle, therefore, as larger numbers of processes execute concurrently, the total time required to calculate results should decrease. This general programming strategy is used for both implementations of the parallel interpolation algorithm which are tested using the following problem: 10,000 control points are used to interpolate a 240 by 800 grid (192,000 cells) using the three closest control points to perform each calculation.

Encore Results

Because the Encore is a multi-user system, small irregularities in execution times can arise from system load variability. To control the effect that multiple users have on overall system performance, we attempted to make timing runs during low-use periods. Given the amount of computation time required for the interpolation experiments that used one and two processors, however, some periods of heavier system use were encountered.

The 10,000-point interpolation problem presents a formidable computational task that is well illustrated by the results contained in Table 1. When a single Encore processor is used, 33.3 hours are required to compute a solution to the problem. The run time is reduced to 2.5 hours, however, when all 14 processors are used. Each Encore processor is not especially fast by today's standards, and, in fact, when a superscalar workstation (RS/6000-550) was used to compute results for the same problem, it was more than four times faster than a single Encore processor. When the full complement of 14 processors is used, however, the advantage of the parallel approach is clearly demonstrated: the Encore is three times faster than the workstation.

Figure 1 shows a monotonic decrease of run times as additional processors are used to compute the interpolated grid. Though the slope of the curve begins to flatten, a greater than order-of-magnitude decrease in computation time is achieved. This indicates that the problem is amenable to parallelism and that further investigation of the problem is warranted.

The results of parallel implementations are often evaluated by comparing parallel timing results to those obtained using a version of the code that uses a single processor. Speedup (Brawer, 1989, p. 75) is the ratio of these two execution times:

$$\text{Speedup} = \frac{\text{Execution Time for One Processor}}{\text{Execution Time for Parallel Version}}$$

Measures of speedup are often used to determine whether additional processors are used effectively by a program. Though the maximum speedup attainable is equal to the number of processors (n) used, speedups are normally less than n because of inefficiencies that results from computational overhead, such as the establishment of parallel processes, and because of inter-processor communication and dependency bottlenecks. Figure 2 shows the speedups obtained for the 10,000-point interpolation problem. The near-linear increase indicates that the problem scales well in the Encore shared-memory MIMD environment.

A measure of efficiency is also sometimes used to evaluate the way in which processors are used by a program. This

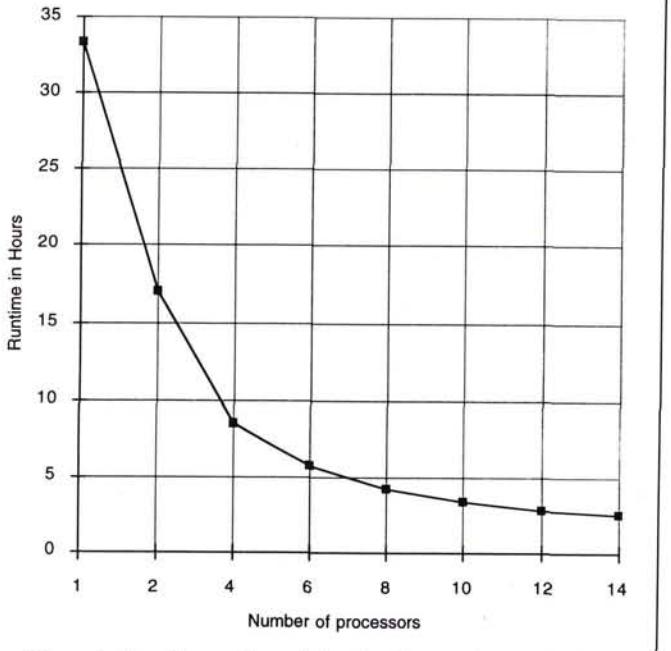

Figure 1. Run times (hours) for the Encore interpolation experiments.

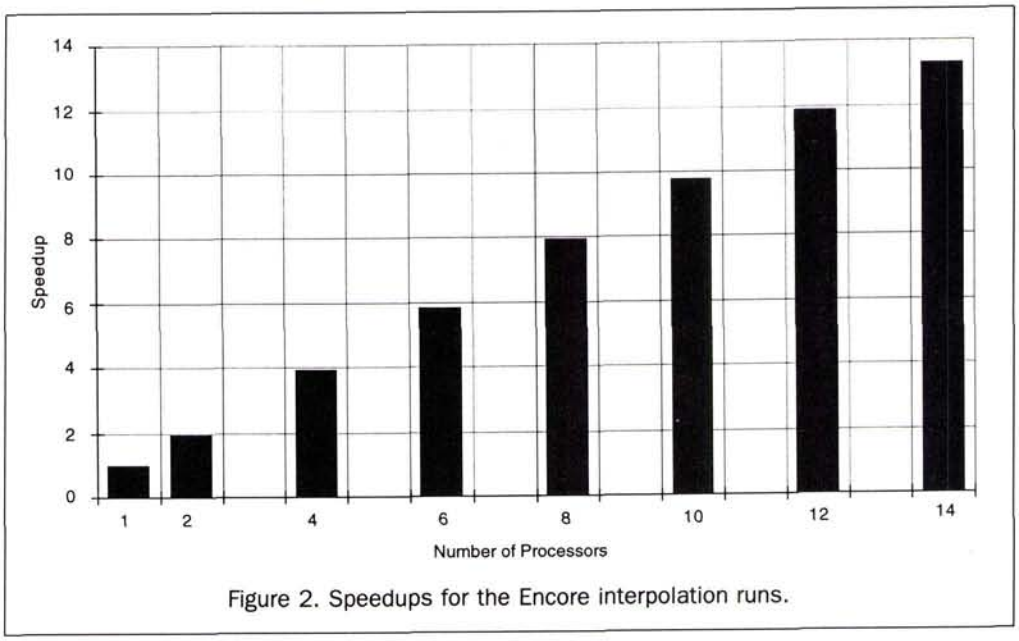

Figure 2. Speedups for the Encore interpolation runs.

TABLE 2. EFFICIENCY OF THE ENCORE INTERPOLATION EXPERIMENTS.

Processors	2	4	6	8	10	12	14
Efficiency	0.99	0.99	0.97	0.99	0.98	0.99	0.95

TABLE 3. KSR1 INTERPOLATION USING THE 240 BY 800 GRID WITH 10,000 POINTS AND THREE-PROXIMAL POINTS.

Processors	Seconds	H:M:S	Speedup	Efficiency
1	7719	2:08:39	1.0	1.00
7	1185	0:19:45	6.5	0.93
15	591	0:09:51	13.1	0.87
58	151	0:02:31	51.1	0.88

measure simply controls for the size of a speedup by dividing it by the number of processors used. If values remain near 1.0 as additional processors are used to compute solutions, then the program scales well. On the other hand, if efficiency values begin to decrease as additional processors are used, then they are being used ineffectively and an alternative approach to decomposing the problem might be pursued. Table 2 shows the computational efficiency of this set of interpolation experiments. The results demonstrate that the processors are being used efficiently across their entire range (2 to 14 processors) because no marked decreases are exhibited. The fluctuations observed can be attributed to the lack of precision of the timing utility (etime) and random variations in the performance of a multi-processor, multi-user system. It is interesting to note, however, that the largest decrease in efficiency occurs as the number of processors is increased from 12 to 14 (the maximum for our Encore environment). It is possible that this decrease is caused, at least in part, by the effect of system-related processes that are factored into the computation of results when all 14 processors are used. Because additional processors are unavailable, however, it cannot be determined if this decrease is caused by the presence of this overhead or by problems associated

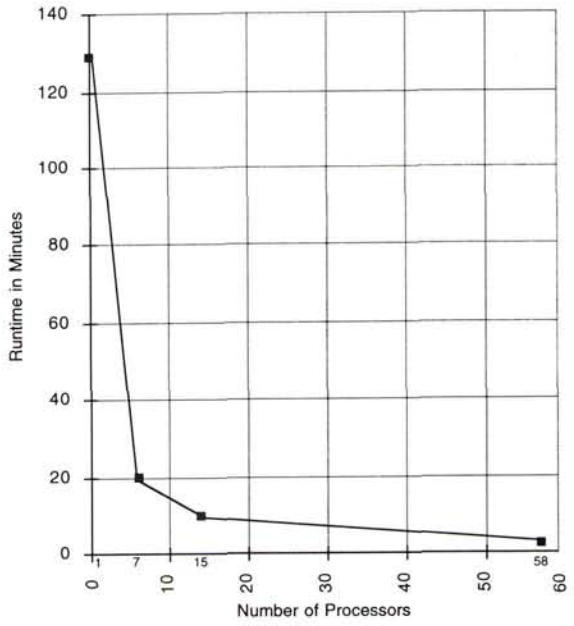

Figure 3. Run times (minutes) for the KSR1 interpolation experiments.

with scalability that are only encountered as larger numbers of processors are added to the computation of results. We therefore wished to determine whether this trend would persist when a larger-scale KRS1 supercomputer with 64 processors was applied to the same problem. We were also interested in using a larger, faster supercomputer because, even when all 14 Encore processors were used, 2.5 hours of computation time were required.

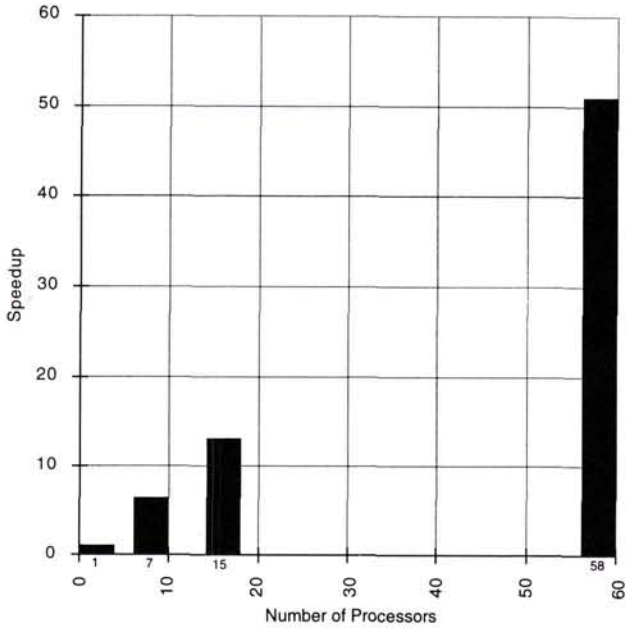

Figure 4. Speedups for the KSR1 interpolation runs.

KSR1 Results

The KSR1 is a dedicated parallel supercomputer. At the time our experiments were conducted, the KSR1 computer at the Cornell Theory Center was configured with 64 processors. However, these processors could only be accessed in a limited number of queues: 1, 7, 15, and 58 processors.

As shown by comparing the results in Table 1 with those in Table 3, each processor in the KSR1 configuration is far faster than those employed by the Encore. When a single processor on each machine was used to compute the same 10,000-point interpolation problem, what took 33 hours on the Encore was reduced to approximately two hours on the KSR1; this decrease is more than an order of magnitude in size. As additional processors are added, computation times continue to decrease dramatically, and when all 58 processors are used, total run time is reduced from more than two hours to two and one-half minutes (Figure 3). The speedup and efficiency measures for the KSR1 implementation also demonstrate that the additional processors are being used effectively. Near-linear speedups are observed (Table 3 and Figure 4) and efficiency remains high (near 0.9) for each of the available configurations of the machine.

Conclusions

Several different approaches to improving the performance of inverse-distance-weighted interpolation algorithms have been developed. One successful approach (Hodgson, 1989) focused on reducing the total number of computations made by efficiently determining those control points that are in the neighborhood of each interpolated cell. When traditional serial computers are used, this approach yields substantial reductions in computation times. The method developed in this paper takes a different tack by decomposing the interpolation problem into a set of sub-problems that can be executed concurrently on shared memory parallel computers. This general approach to reducing computation times will become increasingly commonplace because many computer manufacturers have begun to use parallelism to provide users with cost-effective, high performance computing environments. The approach described here should also work when applied to other computationally intensive methods of interpolation such as kriging (Oliver et al., 1989a; Oliver *et al.*, 1989b) that may not be directly amenable to the neighborhood-search method developed by Hodgson.

The results of our computational experiments show that a parallel, coarse-grained, shared-memory approach to parallel inverse-distance-weighted interpolation yields results that scale well. The measures of efficiency obtained for the Encore implementation are uniformly high: they are greater than or equal to 0.95. The small drop-off in efficiency that we observed in the Encore environment when 14 processors were used did not become substantially greater in the more massively parallel KSR1 environment. In fact, efficiency remained high and fluctuated only slightly when we increased from 15 to 58 KSR1 processors. Our parallel approach to interpolation reduced overall computation times from over 33 hours (2000 minutes) when one Encore processor was used, to approximately two and one-half minutes when the full complement of 58 KSR1 processors was applied to the problem; this represents a speedup of 800. If the RS/6000-550 workstation time is used in place of the Encore as the base for the calculation, the speedup, while more modest at 179, is still quite substantial.

Future research in this area should take place in two veins. The first is to use a more massive approach to parallelism. Alternative architectures such as a heterogeneous network of workstations (Carriero and Gelernter, 1990) or an SIMD machine could be fruitfully investigated. It may be that a more highly parallel brute-force approach can yield performance that is superior to the search-based approaches suggested by Hodgson. The second, and probably ultimately more productive, line of inquiry would meld the work begun here with that of the neighborhood search-based work. The combination of both approaches should yield highly effective results that will transform large, nearly intractable spatial interpolation problems into those that can be solved in seconds.

Acknowledgments

Partial support for this research was provided by the National Science Foundation (SES-9024278). We would like to thank The University of Iowa Weeg Computing Center for providing access to the Encore Multimax computer, and the Cornell National Supercomputing Facility for providing access to the RS/6000-550 workstation and the KSR1 supercomputer. Dan Dwyer, Smart Node Program Coordinator at CNSF, was especially helpful. Michael E. Hodgson, Claire E. Pavlik, Demetrius Rokos, and Gerry Rushton provided helpful comments on an earlier draft of this paper. Amy J. Ruggles created the graphs. This is a revised and expanded version of a paper that appeared in the *Proceedings of Auto-Carto 11.*

References

Armstrong, M.P., and P.J. Densham, 1992. Domain decomposition for parallel processing of spatial problems, *Computers, Environment and Urban Systems*, 16(6):497–513.

Beguelin, A., J. Dongarra, A. Geist, B. Manchek, and V. Sunderam, 1991. *A User's Guide to PVM: Parallel Virtual Machine*, Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, Oak Ridge, Tennessee.

Beguelin, A., J. Dongarra, A. Geist, and V. Sunderam, 1993. Visualization and debugging in a heterogeneous environment, *IEEE Computer*, 26(6):88–95.

Brawer, S., 1989. *Introduction to Parallel Programming*, Academic Press, San Diego, California.

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

Carriero, N., and D. Gelernter, 1990. *How to Write Parallel Programs: A First Course*, The MIT Press, Cambridge, Massachusetts.

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

Cromley, R.G., 1992. *Digital Cartography*, Prentice-Hall, Englewood Cliffs, New Jersey.

El-Rewini, H., T. Lewis, and B. Shriver, 1993. Parallel and distributed systems: From theory to practice, *IEEE Parallel and Distributed Technology*, 1(3):7–11.

Encore Computer, 1988. *Encore Parallel Fortran*, EPF 724-06785, Revision A, Encore Computer Corporation, Marlborough, Massachusetts.

Freund, R.F., and H.J. Siegel, 1993. Heterogeneous processing, *IEEE Computer*, 26(6):13–17.

Hodgson, M.E., 1989. Searching methods for rapid grid interpolation, *The Professional Geographer*, 41(1):51–61.

———, 1992. Sensitivity of spatial interpolation models to parameter variation, *Technical Papers of the ACSM Annual Convention*, American Congress on Surveying and Mapping, Bethesda, Maryland, 2:113-122.

Lam, N-S., 1983. Spatial interpolation methods: A review, *The American Cartographer*, 2:129–149.

MacDougall, E.B., 1976. *Computer Programming for Spatial Problems*, Edward Arnold, London.

———, 1984. Surface mapping with weighted averages in a microcomputer, *Spatial Algorithms for Processing Land Data with a Microcomputer*, Lincoln Institute Monograph #84-2, Lincoln Institute of Land Policy, Cambridge, Massachusetts.

Mitchell, W.J., 1977. *Computer-Aided Architectural Design*, Van Nostrand Reinhold, New York, N.Y.

Monmonier, M.S., 1982. *Computer-Assisted Cartography: Principles and Prospects*, Prentice-Hall, Englewood Cliffs, New Jersey.

Myers, W., 1993. Supercomputing 92 reaches down to the workstation, *IEEE Computer*, 26(1):113–117.

Oliver, M., R. Webster, and J. Gerrard, 1989a. Geostatistics in physical geography. Part I: Theory, *Transactions of the Institute of British Geographers*, 14:259–269.

———, 1989b. Geostatistics in physical geography. Part II: Applications, *Transactions of the Institute of British Geographers*, 14: 270–286.

Pancake, C.M., 1991. Software support for parallel computing: where are we headed? *Communications of the Association for Computing Machinery*, 34(11):52–64.

Shepard, D., 1968. A two dimensional interpolation function for irregularly spaced data, *Harvard Papers in Theoretical Geography*, Geography and the Property of Surfaces Series No. 15, Harvard University Laboratory for Computer Graphics and Spatial Analysis, Cambridge, Massachusetts.

White, D., 1984. Comments on surface mapping with weighted averages in a microcomputer, *Spatial Algorithms for Processing Land Data with a Microcomputer*, Lincoln Institute Monograph #84-2, Lincoln Institute of Land Policy, Cambridge, Massachusetts.

Marc Armstrong
Marc Armstrong has a B.A. degree from SUNY-Plattsburgh, an M.A. degree from UNC-Charlotte, and a Ph.D. degree from the University of Illinois at Urbana-Champaign. He is currently an Associate Professor at The University of Iowa where he holds appointments in the Departments of Geography and Computer Science and in the Program in Applied Mathematical and Computational Sciences. His research focuses on the design of spatial decision support systems and methods of spatial analysis.

Richard Marciano
Richard Marciano received the M.S. and Ph.D. degrees in computer science from the University of Iowa in 1989 and 1992, respectively. For the past six years he has been a supercomputer and parallel processing consultant. His professional interests include computational science, parallelizing compilers, algebraic compiler environments, and high-speed computing. He is currently a Computational Scientist at the National Supercomputer Center for Energy and the Environment, University of Nevada, Las Vegas.

Call for Nominations
ASPRS Fellow and Honorary Member Awards

ASPRS would like to encourage members to submit nominations for its Fellow and Honorary Member awards.

The **Fellow Award** is relatively new. It is conferred on active Society members who have performed exceptional service in advancing the science and use of the mapping sciences and for service to the Society. Nominees must have been active members of the Society for at least ten years at the time of their nomination.

The **Honorary Member Award** is ASPRS's highest honor. It recognizes an individual who has rendered distinguished service to the Society and/or who has attained distinction in advancing the science and use of the mapping sciences. It is awarded for professional excellence and service to the Society. Nominees must have been active members of the Society for at least fifteen years at the time of their nomination.

For more information, or to obtain a nomination form for these very special awards, please call Diana Ripple at 301-493-0290.