Interpolation of spatially inhomogenous data sets: An evaluation of parallel computation approaches

Barton E. Cramer
University of Iowa

Marc P. Armstrong
University of Iowa

Copyright © 1997 Barton E. Cramer and Marc P. Armstrong

INTERPOLATION OF SPATIALLY INHOMOGENEOUS DATA SETS: AN EVALUATION OF PARALLEL COMPUTATION APPROACHES

Barton E. Cramer

and

Marc P. Armstrong

Department of Geography
316 Jessup Hall
The University of Iowa
Iowa City, IA 52242 USA
(319) 335-0150
barton-cramer@uiowa.edu

ABSTRACT:

Parallel processing hardware is becoming increasingly affordable, with 4-processor, Pentium-based servers near the $10,000 price point. Prior research has shown that a number of routine GIS spatial analysis tasks can be performed in parallel with high efficiency. The purpose of this paper is to investigate one such application, the parallel computation of interpolated surfaces using the neighborhood-based, inverse-distance-weighted Clarke algorithm, and to examine the effects of different distributions of control points on compute time. These data distributions include: evenly spaced, random, loosely clustered, and tightly clustered (as measured by the nearest neighbor statistic). Following a review of non-parallel performance characteristics of the Clarke algorithm, the programming scheme used to achieve parallelism is outlined, and two strategies for subdividing the spatial interpolation task among the parallel processes are described. Tests carried out on a Silicon Graphics Power Challenge computer show that while parallel interpolation is effective at reducing run time, the choice of domain decomposition strategy significantly impacts the amount of reduction when processing clustered data. Our results point to a generic strategy that can be used to apportion spatially inhomogeneous tasks in multiprocessing environments.

1. INTRODUCTION

Parallel processing has been shown to reduce the elapsed (run) time required for a variety of geographic information system (GIS) or spatial decision support system (SDSS) tasks, such as nearest-neighbor analysis (Cramer & Armstrong, 1996), line simplification (Mower, 1996), computing spatial association measures (Armstrong, Pavlik & Marciano, 1994; Li, 1996; Rokos & Armstrong, 1996), census polygon classification (Openshaw & Turton, 1996), zonal aggregation (Openshaw & Schmidt, 1996), network shortest path determination (Ding, Densham & Armstrong, 1992), "ridding (Armstrong & Marciano, 1994; Hodgson et al., 1995), terrain feature extraction (Rokos & Armstrong, forthcoming), and terrain visualization (Shickar et al., 1996). The relevance of parallel processing research is reinforced by steady reductions in the market cost of hardware, the growing availability of low-end (typically four-processor) parallel computers, and by ongoing efforts to harness "spare cycles" from already deployed workstations and high-end PCs through the use of local or wide-area networked (distributed) approaches to achieving parallelism (sometimes referred to as networks of workstations, or NOW: (Anderson, Culler & Patterson, 1995)).

The application investigated in this paper is the interpolation of a regularly spaced x-y grid from a set of (x, y, z) points that are presumed to be scattered samples gathered from a continuous surface (e.g., elevation). The interpolation code that has been used is a modification of a demonstration, non-parallel, C-language program published by Clarke (Clarke, 1995). The serial algorithm used by Clarke is already considerably more efficient than a naive, "brute force" approach to finding the nearest-neighbors on which interpolated values are based (Armstrong & Marciano, 1994). Like the "brute force" algorithm, however, Clarke's approach is readily decomposed into independent parts to take advantage of multiple, simultaneous (i.e., parallel) processes. The actual performance enhancements achieved through the use of parallel processing are documented, based on four data sets specifically chosen to exhibit a spectrum of location patterns ranging from evenly dispersed to tightly clustered. As data clustering increases, it is seen that run times for the Clarke algorithm escalate rapidly, but parallel processing always realizes a significant improvement over serial (single-processor) performance. Choosing how to subdivide the processing work among the parallel processors (domain decomposition) is investigated by comparing two schemes having both significantly different basic task size (granularity), and spatial distribution of task assignments. The finer-grained and more spatially dispersed decomposition is shown to be generally superior for a shared memory architecture, suggesting a number of directions for further research.
2. DATA

The (x, y, z) point data sets needed to test grid interpolation were created using specially developed programs that can generate (1) an approximately uniform data set; (2) a random x-y pattern of uniform (average) density; (3) a pattern of random clusters of randomly scattered points with a specific spatial density; or (4) any combination of (2) and (3). Figure 1 shows an evenly dispersed data set, together with the random, loosely clustered, and tightly clustered data sets that were used to compute the results reported in this paper. The random and clustered data sets each contain exactly 1000 points, however, due to the geometric pattern required with the evenly dispersed data set, it necessarily contains 1013 points. The degree of spatial clustering can be measured quantitatively, for example by use of the nearest neighbor index $R$, which ranges from 0.0 for completely clustered data, through 1.0 for randomly distributed data, and up to 2.15 for evenly dispersed data (e.g., see Ebdon (1985), DO. 143-148). Values of $R$ for each test data set are given in Figure 1.

The spatial distribution of the input data has a significant impact on the performance of the Clarke algorithm, as will be seen in the test results. The data z-values have no impact on the Clarke algorithm, however, so test data set z-values were taken from a gently sloping plane $z = 0.1 (x + y)$, to facilitate output evaluation and code debugging.

3. HARDWARE / SYSTEM ENVIRONMENT
All parallel processing experiments reported in this paper were conducted using a Silicon Graphics (SGI) Power Challenge XL system, running under IRIX 6.2 (a version of the UNIX operating system). The system includes sixteen individual processing elements (CPUs): ten operate at 90 MHz and six operate at 75 MHz. This means that ten of the processors are capable of performing 1.2 times as many computations a given time interval as each of the other six. This modest heterogeneity in the computing environment can have an effect on parallel workload balancing because the variability of performance makes it more difficult to ensure that all the processors assigned to a job are kept constantly busy.

The Power Challenge is a shared memory multiprocessor, meaning that each individual processor can access all of the machine's 2 gigabytes (GB) of main memory. This architecture is also referred to as a symmetric multiprocessor (SMP), or sometimes as a "tightly coupled" or "shared everything" system (Thompson, 1996, p. 52). Under this approach, all processors can access a copy of any input data set if it is placed in shared memory. In the evaluations presented in this paper, a system-specific, C-language implementation exploited the Power Challenge's shared memory architecture to realize data storage and message-passing efficiencies. Other operating environments and architectures could require alternative approaches to parallelization, and while we believe our results to be generally applicable, caution should be exercised when inferring performance on other systems. In addition, although as much of the testing as possible was carried out while the Power Challenge was reserved for our exclusive use, some tests were necessarily performed while there was other usage of the system. Minor fluctuations in recorded performance may be attributable to incidental timesharing of processors, or to occasional contention with other users for system resources.

4. CLARKE ALGORITHM

The interpolation algorithm presented by Clarke (Clarke, 1995, pp. 250-251) has four principal steps:

1. Setup: Read an (x, y, z) data point file and set the number of rows and columns to be generated (which directly implies grid cell x and y dimensions).
2. Initialization: Place the z-value of each input data point into the grid cell in which it falls (many cells will typically not be initialized).
3. Interpolation: Use only the initialized cells to interpolate all remaining non-initialized cells.
4. Output: Output the completed grid to a file or display (as desired), and report results to the user.

Steps one and four (setup and output) typically consume only a very small fraction of total run time (on the order of 0.2-2%). Step two (initialization) also consumes relatively little time unless the volume of input data is equivalent to, or larger than, the number of grid cells to be computed. For most jobs, step three (interpolation) is by far the most computationally intensive and time-consuming, and, as a consequence, this is where our parallelization efforts have been concentrated.

Neighborhood-based, inverse-distance weighted interpolation generates a z-value for any location by using the k closest neighboring data points. The z-value of any desired point is computed using the formula

\[ Z_p = \frac{\sum_{i=1}^{k} \frac{z_i}{d_i^\beta}}{\sum_{i=1}^{k} 1/d_i^\beta} \]

where:

- \( Z_p \) = interpolated value at point \( p \),
- \( z_i \) = observed value at point \( i \), in the neighborhood of \( p \),
- \( k \) = number of points in the neighborhood of \( p \) that are used in the interpolation (often \( k < 10 \)),
- \( d_i \) = Euclidian distance from point \( i \) to \( p \), and
- \( \beta \) = distance weighting factor (often \( \beta = 2 \)).

Determining nearest neighbors in Clarke's initialized grid could be accomplished through an exhaustive search, in which every initialized cell is compared with the cell to be interpolated, in order to find the k closest. Instead, Clarke accelerates this process
by taking advantage of the grid topology, and working outwards from the cell to be interpolated in a series of concentric search squares (Figure 2).

![Clarke grid search method.](image)

This process only requires examining each successive square of cells until at least the required number of initialized cells \(k\) is located. When at least \(k\) cells have been found, the search is terminated. Obviously, the greater the number of initialized cells that are nearby, the sooner the search process will end, implying that initial control point distribution will have an impact on algorithm performance, with clustered data requiring more searching and thus taking more time. This can be seen clearly in the timing results presented in Section 6.

5. PARALLEL IMPLEMENTATION

The general parallelization approach used to address the interpolation problem is shown in Figure 3. Note that a single process (referred to here as the master) handles the relatively short setup, initialization and output steps, while all processes (including the master) share in the time-consuming interpolation step. In other words, a strategy of partial parallelization has been followed (Morton & Tyler, 1996). Such a strategy seeks to minimize programming effort and maximize parallel performance gains.

![C-function parallel processing flow.](image)

The other relevant aspect of parallel program design is the subdivision of the overall processing task into the parts that will be assigned to each processor. This is referred to as domain decomposition, and it generally has the objective of balancing the workload.
among processors to avoid the loss of efficiency caused by idle processors (e.g., processors that finish early). Two domain decomposition strategies were evaluated:

- **jumpRow**: Each process starts on a defined row (process 1 starts on row #1, process 2 starts on row #2, etc.), and then "leapfrogs" through the grid, advancing by a number of rows equal to the number of active processes.
- **seqRows**: The grid is divided into \( n \) strips of adjacent rows, according to the number of processes \( n \). Each process is thus assigned an approximately equal-sized strip of spatially contiguous rows to interpolate.

While both of the above strategies result in approximately equal numbers of rows being assigned to each process (if the number of rows does not divide evenly by the number of processes, some processes will have an additional row to interpolate), in the case of jumpRow, the rows are taken from various parts of the grid, but for seqRows, the rows are contiguous. Note, however, that these two approaches guarantee equal processor work loads only if the input data are evenly spatially distributed.

### 6. PARALLEL CODE RESULTS

The two domain decomposition strategies described in Section 5 were each implemented in C-code, and tested using the four test data sets (Figure 1). The Power Challenge operating system was allowed to assign the processors for each run, which it does apparently at random, usually resulting in a mix of fast and slow CPUs, but sometimes resulting in all fast or all slow CPUs being chosen. Such a variation in processor mix will lead to execution time variation between tests, so all results in this section are averages of five runs.

Parallel results are presented directly as run times, and in the form of speedup. Speedup is defined as the ratio of the run time realized on a single processor to that taken with \( n \) parallel processors, and gives a unitless measure that can be more easily compared across alternative approaches. For all results presented below, speedup is calculated relative to serial code running on a single 90MHz processor.

Table 1 and Figure 4 shows the run times for the jumpRow decomposition scheme, using each of the four test data sets. Figure 5 shows the results of Table 1 presented as speedup. This graph confirms the relative performance observed in the raw run times, and also shows desirable, steadily increasing, speedups (indicating scalability) for all options. The higher speedups realized with the clustered data are attributed to the greater relative proportion of program time spent in the compute-intensive interpolation step, where the impact of multiple processors is felt. Note, that speedup is significantly below the theoretical (ideal) curve, and gradually falls off in slope. As mentioned earlier, all speedup values have been computed by taking serial C-code running on a fast (90MHz) processor for the base-case time. In contrast to typical practice, the theoretical speedup curves shown here have a decreased (less than unity) slope above 10 processors. This is because it is arbitrary assumed that only 90MHz processors will accessed at first (up to and including processor 10), followed by the somewhat less powerful 75MHz processors (from 11 through 16). Table 2 and Figures 6 and 7 show similar results for the seqRows decomposition strategy, and generally similar observations apply. In addition, it can be seen that for the uniform and random data sets, the run times for the jumpRow domain decomposition are essentially the same as those for seqRows, while for the more time-consuming clustered data sets, jumpRow is typically faster. For example, with loosely clustered data and 8 processors, seqRows takes 1.65 times as much run time as jumpRow (4.37 sec / 2.64 sec).
<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Uniform</th>
<th>Random</th>
<th>Loosely clustered</th>
<th>Tightly clustered</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.05</td>
<td>6.40</td>
<td>15.75</td>
<td>41.08</td>
</tr>
<tr>
<td>2</td>
<td>3.83</td>
<td>3.84</td>
<td>8.29</td>
<td>22.56</td>
</tr>
<tr>
<td>3</td>
<td>2.77</td>
<td>2.63</td>
<td>6.29</td>
<td>15.18</td>
</tr>
<tr>
<td>4</td>
<td>2.12</td>
<td>2.25</td>
<td>4.87</td>
<td>12.24</td>
</tr>
<tr>
<td>5</td>
<td>1.86</td>
<td>1.92</td>
<td>3.97</td>
<td>10.04</td>
</tr>
<tr>
<td>6</td>
<td>1.70</td>
<td>1.75</td>
<td>3.49</td>
<td>8.50</td>
</tr>
<tr>
<td>7</td>
<td>1.46</td>
<td>1.63</td>
<td>2.97</td>
<td>7.26</td>
</tr>
<tr>
<td>8</td>
<td>1.37</td>
<td>1.42</td>
<td>2.64</td>
<td>6.45</td>
</tr>
<tr>
<td>9</td>
<td>1.28</td>
<td>1.34</td>
<td>2.43</td>
<td>6.08</td>
</tr>
<tr>
<td>10</td>
<td>1.22</td>
<td>1.23</td>
<td>2.26</td>
<td>5.36</td>
</tr>
<tr>
<td>11</td>
<td>1.17</td>
<td>1.24</td>
<td>2.19</td>
<td>5.08</td>
</tr>
<tr>
<td>12</td>
<td>1.08</td>
<td>1.15</td>
<td>2.03</td>
<td>4.64</td>
</tr>
<tr>
<td>13</td>
<td>1.11</td>
<td>1.05</td>
<td>1.91</td>
<td>4.40</td>
</tr>
<tr>
<td>14</td>
<td>1.02</td>
<td>1.09</td>
<td>1.86</td>
<td>4.29</td>
</tr>
<tr>
<td>15</td>
<td>1.08</td>
<td>1.02</td>
<td>1.77</td>
<td>3.91</td>
</tr>
<tr>
<td>16</td>
<td>1.01</td>
<td>1.07</td>
<td>1.72</td>
<td>3.77</td>
</tr>
</tbody>
</table>

Table 1. Run times in seconds for four test data sets using \texttt{jumpRow} decomposition (200 x 200 grid; \(k = 3\)).

![Graph](image)

Figure 4. Run times for \texttt{jumpRow} decomposition strategy (200 x 200 grid; \(k = 3\)).
Figure 5. Speedup for jumpRow decomposition strategy (200 x 200 grid; $k = 3$).

Table 2. Run times in seconds for four test data sets using seqRows decomposition (200 x 200 grid; $k = 3$).
Figure 6. Run times for \texttt{seqRows} decomposition strategy (200 x 200 grid; $k = 3$).

Figure 7. Speedup for \texttt{seqRows} decomposition strategy (200 x 200 grid; $k = 3$).
Overall, these results may be interpreted as showing that:

- Useful performance improvement is achieved from parallelization when using any of the four test data sets.
- Data clustering substantially increases algorithm run times.
- The impact of clustering is most strongly demonstrated when the amount of clustering is high.
- The `jumpRow` decomposition strategy typically outperforms the `seqRows` strategy for the more time consuming clustered data sets.

7. CONCLUDING DISCUSSION

The primary goal of the parallel programming research reported in this paper was to investigate the effects of inhomogeneous data distribution on run-time performance of a parallel grid interpolation application (the Clarke algorithm). Parallel conversion of Clarke’s serial code shows that it can be put into partially parallel form, and that useful improvements to performance can be realized through a partial parallelization. The comparison of two parallel processing domain decomposition strategies shows that the choice of domain decomposition strategy can influence parallel performance results, with the medium-grained and spatially distributed `jumpRow` strategy realizing consistently lower run times than the coarser-grained and spatially contiguous `seqRows` strategy on the more difficult to interpolate clustered data sets.

The results obtained in this paper suggest several opportunities for both practical application and further investigation:

1. Establishment of an optimum level of granularity for a suite of typical data distributions by starting with a single cell and progressing systematically (e.g., to 2, 3, 4, ..., adjacent cells).
2. Investigation of other options for sampling the work-space domain.
3. Investigation of the importance of the parallel / serial processing time ratio, for example by substituting kriging or another more compute-intensive interpolation method for the inverse distance weighted average.

In the longer term, the execution of parallel applications such as spatial interpolation in a networked workstation environment would also permit the analysis of (1) performance of parallelization alternatives across a network, where message passing becomes a much more time-consuming factor, and (2) the impact on practical applications of a more heterogeneous collection of processors.

ACKNOWLEDGEMENTS

Partial support for this work was provided by the National Science Foundation’s Division of Undergraduate Education through grant DUE-9551183. Computing resources were provided through The University of Iowa Advanced Research Computing Services (ARCS) group. The advice and assistance of ARCS graduate assistants Tom Halverson and Eric Van Wyk is gratefully acknowledged.

REFERENCES


