Experiments in the Concurrent Computation of Spatial Association on the KSR-1

Richard J. Marciano[1]
Marc P. Armstrong[2]
Claire E. Pavlik[3]

January 10, 1993

richard-marciano@uiowa.edu
Fencing Technologies
phone & fax: (612) 926-7955

marc-armstrong@uiowa.edu
claire-pavlik@uiowa.edu
Department of Geography
phone: (319) 335-1050
fax: (319) 335-2725

1. Gerard P. Weeg Computer Center, University of Iowa, and Fencing Technologies, Minneapolis, Minnesota.

2. Departments of Geography and Computer Science, and Program in Applied Mathematical and Computational Sciences, University of Iowa.

3. Department of Geography, University of Iowa.

Acknowledgments: This work was supported in part by the National Science Foundation (SES-9024278). The Cornell National Supercomputing Facility provided access to the KSR-1 computer system that was used to prepare the results reported in this paper. We wish to thank Arthur Getis for kindly providing the original serial version of the code used in the analyses. Many thanks to Dan Dwyer, former Smart Node Program Coordinator at Cornell, and to Mike Hammill, current Smart Node Program Coordinator, for their technical insights and parallel programming assistance.

TABLE OF CONTENTS

Abstract..... **1**

1. INTRODUCTION..... **1**

1.1 Background: G Statistic..... 2

1.2 Computational Requirements..... 3

2. THE KSR-1 ENVIRONMENT..... **5**

2.1 Hardware Description..... 6

2.2 Parallel Programming Software Support..... 7

3. IMPLEMENTATIONS USING A SINGLE PROCESSOR..... **8**

3.1 Serial Algorithm Modifications..... 10

3.2 Results: Single Processor..... 11

4. PARALLEL IMPLEMENTATIONS..... **14**

4.1 Parallel Algorithm Modifications..... 14

4.2 Parallel "Naive" Results..... 19

4.3 Parallel "Sorting" Results..... 25

4.4 Comparison of Parallel "Sorting" with Serial "Naive"..... 26

4.5 Comparison of the Parallel Implementations..... 28

5. CONCLUSIONS..... **29**

References..... **30**

Experiments in the Concurrent Computation of Spatial Association on the KSR-1

Abstract

Spatial association measures, when computed for large datasets, have significant computational requirements. Parallel processing is one way to address these requirements. The design of parallel programs, however, requires careful planning since many factors under programmer control affect the efficiency of the resulting computations. In this paper, two strategies of parallel processing are described using as an illustration a measure of spatial association, **G**(d). An evaluation is made of the efficiency of the parallel implementations by varying the problem size and the number of processors used in parallel. The results obtained indicate that parallel processing can currently enable analysts to work in near-real-time with problems that range in the tens of thousands of observations; such problems require less than two minutes of execution time on the KSR-1. Also superlinear speedups of almost 1500 are obtained for the *computation phase* of the problem.

Key Words: Parallel processing, spatial statistics, KSR, MPP

1.0 INTRODUCTION

The need to incorporate spatial dependencies in geographical models has a long history of acknowledgment in theory, but neglect in practice. Gould (1973), for example, questioned the application of classical inferential statistical methods to geographical data, while Cliff and Ord (1973) demonstrated that spatial dependencies, with their underlying assumption of independent observations, can throw the interpretation of regression results into question. Though Anselin and Griffith (1988) and Getis and Ord (1992), among others, have recently refocussed attention on the measurement of spatial dependency, for the past two decades practitioners have often ignored its measurement for several reasons:

(1) Researchers have often lacked the training needed to perform spatial dependency analyses. As a result of the recent publication of tutorials (e.g. Goodchild, 1986; Griffith, 1987; Odland, 1988) and more comprehensive texts on spatial statistics (e.g. Cressie, 1992), this problem is becoming less significant.

(2) Easy to use software that evaluates datasets for spatial dependencies has not been widely available. This problem is also becoming less significant as researchers have demonstrated that commercial statistical software can be applied to geographical problems (Griffith, 1987) and have developed programs that are specifically designed to evaluate data for spatial dependencies (Anselin, 1991). Moreover, many GIS software packages now provide a means for measuring spatial dependencies.

(3) The application of many measures of spatial dependence to large realistic problems has not been feasible because of the numerical intensity of the required computations (e.g. Griffith, 1990).

The purpose of this paper is to address this third issue. We illustrate the magnitude of these computational requirements and present the results of a series of computational experiments that were developed to reduce execution times when a recently derived (Getis and Ord, 1992) measure of spatial dependency, **G**(d), is applied to large datasets. The experiments are designed to show how a straightforward translation of statistical equations into computer code can be improved by decomposing the problem into a set of processes that are executed concurrently using a parallel supercomputer.

1.1 Background: G Statistic

G(d) measures the spatial association of positive-valued point data in the plane; it does not provide a single, global assessment of spatial structure but, instead, is reported as a series of measures calculated for user-specified distances. Getis and Ord (1992) state that this intra-zonal focus is important when data have unequal local averages and when values are related by the distance between observations rather than by absolute location. These characteristics are often found in spatial data collections (Cliff and Ord, 1981; Oliver *et al.*, 1989).

G(d) can be summarized in the following way: Define a dataset of *n* observations each described by coordinates and a positive, real-valued variable, **Z**. Since the **G**(d) statistic summarizes the characteristics of the **Z**-values contained in circles of varying sizes, select an observation, located at site *i*, and center a circle of radius *d* upon it. Compute the sum of the products of site *i*'s **Z**-value with the **Z**-value of the other sites within the circle, and accumulate these products. Repeat the procedure for each site using circles of the same radius, then sum the accumulated values for all sites. Finally, normalize this result by dividing it by the double-sum of all different site $z_i \cdot z_j$-products. Repeat the process for all specified radius values.

1.2 Computational Requirements

The calculation of **G**(d) requires two main steps, referred to below as *initialization* and *computation*, that each requires substantial amounts of computation. In the *initialization phase*, vectors X, Y, Z are read, interpoint distances are computed and the results are stored in a matrix (called "DIST").

```
      DO 2 J=1,NOBS
            DO 2 I=1,NOBS
                  IF (I .NE. J) THEN
                        XDIST = X(I) - X(J)
                        YDIST = Y(I) - Y(J)
                        DIST(I,J) = SQRT ((XDIST * XDIST) + (YDIST * YDIST))
                  END IF
2     CONTINUE
```

The number of calculated distances increases with the square of the number of observations in this example, though simple arithmetic assignments can reduce the amount of computation in this part of the program provided that DIST(I,J) = DIST(J,I). Thus for a 10,000 observation analysis, approximately 10^8 distance calculations (N^2-N) are required. If the symmetry of the matrix is exploited to reduce computation 49,995,000 distances (N^2-N)/2) must be computed.

In the *computation stage*, for each d-value, DIST is used to ensure that only points that fall within a window specified by the current value of the radius are included in the computation of **G**(d). This process is repeated until the radius reaches the desired maximum value of *d*. Thus, the total amount of computation required depends partly on the number of radii for which **G**(d) is computed. In general, the *computation time* of

this phase increases as a function of the size of the problem and the number of radii for which **G**(d) is computed.

The following fragment of code illustrates the computation stage where d is called STEP, and where a G value corresponding to that particular STEP value is computed (G = ZS / ZSUM):

```
      STEP = 0.
70    STEP = STEP + (MAX / INC)
            ZSUM = 0.
            ZS = 0.
            WT = 0.
            DO 3 J=1,NOBS
                  WA = 0
                  DO 3 I=1,NOBS
                        IF (I .NE. J) THEN
                              ZSUM = ZSUM + (Z(I) * Z(J))
                              IF (DIST(I,J) .LE. STEP) THEN
                                    ZS = ZS + (Z(I) * Z(J))
                                    WT = 1 + WT
                                    WA = WA + 1
                                    W(J) = WA
                              END IF
                        END IF
3           CONTINUE
            G = ZS / ZSUM
            S2 = 0.
            DO 4 J=1,NOBS
                  S2 = S2 + W(J)*4*W(J)
4           CONTINUE

C           ..... MORE CODE LEFT OUT .....

      IF (STEP .EQ. MAX) GO TO 80
      GO TO 70
80    CONTINUE
```

To demonstrate its computational requirements, a serial implementation of code that calculates **G**(d) was ported to three platforms: a **PC** (20 Mhz IBM PS2/55 SX 386 processor with no floating-point accelerator), a **mainframe** (Encore Multimax), and a **supercomputer** (KSR-1). Two important limitations were examined: the maximum problem size and runtime.

With 256 observations, the executable code generated on the PC used close to 300KB; when the number of observations was increased to 384, the PC's memory capacity (640KB) was exceeded. Consequently, the largest problem size that could be run on a PC was approximately 300 points. When the problem size was reduced to 256

observations and **G**(d) was calculated for 20 radii increasing in increments of 5 units from 5 to 100, the PC required nearly 20 minutes to calculate the results (Table 1). While faster PCs are available and memory restrictions are being overcome with the provision of new operating systems, PC-class computers preclude interactive analyses for the **G**(d) statistic and limit the size of the analyses that may be successfully undertaken.

Table 1. Execution times (seconds) for a 256 point data set.

	386 PC	**Encore**	**KSR-1**
Initialization	87.0	3.60	0.04
Computation	1106.0	29.30	1.62
Total Time	**1193.0**	**32.90**	**1.66**

As one might expect, the mainframe class Encore computer is capable of handling larger datasets. The largest sized problem that could be accommodated, however, is still fairly small with 1600 points. Also note from Table 1 that the Encore required more than half a minute of execution time for the small 256 point dataset, while the total runtime for the KSR-1 was less than two seconds. These preliminary analyses demonstrate that high performance computing is required for two reasons: first, to handle large problems (we would like to analyze data sets that contain more than 10,000 observations) and, second, to obtain runtimes that enable analysts to perform interactive exploratory analyses of large spatial data sets. Consequently, all further analyses are reported using the KSR-1 parallel supercomputer. In section 2, we briefly describe the characteristics of this high performance computer. Following that, we describe two different implementations of the **G**(d) algorithm and evaluate their performance in a series of processing experiments that exploit the architectural characteristics of the KSR-1.

2.0 THE KSR-1 ENVIRONMENT

The KSR-1 is a massively parallel computer composed of 64-bit RISC-style superscalar processors with independent integer and floating point units; the floating point units also have independent, pipelined adder/multipliers. Each processor used by the KSR-1 is called a cell and runs at 20-MHz for a peak performance rate of 40 Megaflops per cell; the number of cells may range from 8 to 1088. The model we used

at the Cornell National Supercomputer Facility was a 64-cell machine, structured as a ring of 2 rings of 32 cells each (see section 2.1 for details).

2.1 Hardware Description

The KSR-1 is a hybrid architecture in the sense that both the shared-memory and distributed-memory computing models are used. At first glance, it may seem to be a shared-memory machine in which programs and data are mapped into a single large virtual address space. However, physical storage is composed of a collection of local caches, one for each processor (or cell), each capable of storing 32 MBytes of information. Each cell also has an additional, faster 0.5 MB hardware cache consisting of a 256KB data cache, called the subcache, and a 256KB instruction cache. Data is stored in units of pages and subpages; a local cache can store 2K pages, each divided into 128 subpages of 128 bytes (2K x 128 x 128 = 32MB).

Local caches are interconnected through a mechanism called the search engine. If a cell must access memory located outside its own cache, it generates a request to a search engine that interconnects the local caches and provides routing and directory services for all caches. It is this characteristic of the architecture that lets the distributed local caches behave logically as a single, shared address space. The combination of local caches and the search engine is implemented as a hierarchy of search groups. If a cell requests access to a memory location that is not in its subcache, then the search engine attempts to fetch data from the local cache. If the required data are not in the local cache, then the search engine generates a request that is placed into an open slot on the local communications ring (Ring:0). This request (a packet) is passed to the next cell on the ring, which then tries to match the packet against its local cache by consulting the cell's local-cache directory. If the match is successful, the packet is extracted from the ring, reinserted with the data, and continues around the ring back to the cell that generated the request. One of the Ring:0 cells is a special cell that connects to the next higher level (Ring:1) in the hierarchy. This cell includes a directory comprising all the entries in Ring:0's local-cache directories. When a packet reaches this cell, if the requested address is not present in the Ring:0 directory, then the packet is routed to the next higher level, Ring:1, which connects up to 34 other Ring:0s.

The aggregate capacity of each level of the cache hierarchy and the latency (in processor cycles) between the processor and each cache level is shown in Table 2.

Latency times are very important in this architecture because when a cell generates a memory request, processing cannot proceed until the needed data arrives. Given that latencies increase when non-local memory references are encountered, to be efficient programs must minimize access to memory locations that are far, in an architectural sense, from the local cache. Because performance is linked to an effective exploitation of the cache hierarchy, the characteristics of a problem must be matched to the characteristics of the hardware for maximum performance.

Table 2. Cache hierarchy characterization.

Location of cache	*Local cache capacity*	*Latency-clock cycles*
Local subcache	256KB	2
Local cache	32MB	18
Search Group:0	1GB	175
Search Group:1	34GB	600
Page fault	N.A.	400,000 (~20 msecs)

Source: Kendall Square Research, 1992b.

2.2 Parallel Programming Software Support

The underlying mechanism used to execute parallel constructs in this programming environment is a pthread, a single sequential flow of control within a process (Kendall Square Research, 1992a). A FORTRAN program creates, synchronizes, schedules, and cancels pthreads using library calls or compiler directives that invoke calls to the runtime library. There are four main high level FORTRAN parallel constructs that are normally used to implement programs on the KSR-1:

(1) Parallel regions -- multiple instantiations of a code segment are executed in parallel.

(2) Parallel sections -- multiple code segments are executed in parallel.

(3) Tile families -- loops are executed in parallel (often used with affinity regions). Loops can be tiled automatically, semi-automatically, or manually by the programmer.

(4) Affinity regions -- all tile families in an affinity region are executed by the same team of pthreads, using the same pthread-to-tile assignments. This ensures that data movement is minimized and cache performance is improved.

These constructs are implemented using compiler directives that start with **C*KSR*** to improve portability and readability of code. *Tile,* for example, is a commonly used directive that divides loop iterations into blocks that run on different threads. The tiling strategy used by a parallel program affects the partitioning and assignment of the workload to the available processors. The tile directive has the following syntax:

c*ksr* tile (**index_list** [options])
..... loop to be tiled .....
c*ksr* end tile

The options used in our parallel spatial statistics implementation are:

- index_list
specifies the index variables of the loops to be tiled.

- private = (variable_list)
specifies variables for which each thread has its own "private" copy.

- reduction = (variable_list)
specifies a list of variables which are reduction variables for the tile family. Reduction variables accumulate values for separate tiles that need to be pooled into a single value at the end of the parallel construct.

- strategy = (slice | mod | grab | wave)
specifies the strategy to be used to order the execution of tiles in a tile family. We used the **slice** strategy in which loop iterations are divided into a number of tiles equal to the number of threads executing the tile family. Each pthread executes one tile, and each tile is the same size. This facilitates load balancing among the processors.

- teamid = (team_id)
specifies the ID of the team used to execute the tile family.

3.0 IMPLEMENTATIONS USING A SINGLE PROCESSOR

Two implementations of **G**(d) are investigated in this section. The first, a "naive" implementation, is a straightforward translation of the equations reported by Getis and Ord (1992) into a series of FORTRAN statements that are executed using one processor. This serves as a serial benchmark against which the parallel results reported in the

following section can be compared. The second, what we call a "sorting" implementation, is intended to improve performance by restructuring the way that statistical indices are calculated.

3.1 Serial Algorithm Modifications

In the "sorting" modification, the structure of the problem is exploited: distance values in each column of the matrix, DIST, are sorted in ascending order using a *shellsort* algorithm given in Sedgewick (1990), and a matrix of *zi*•*zj* products, ZZ, is sorted into the same order by columns. The distance matrix, DIST, is then used to construct a "mask" of the *zi*•*zj* products that fall within the initial d-radius and successive, incremental rings about each data point.

X	40.8	77.9	25.9	42.1	53.1	4.0	58.6	53.2	39.7
Y	66.7	23.5	48.8	34.6	91.5	49.2	30.8	27.5	57.9
Z	74.7	30.6	62.6	78.6	25.0	18.4	70.1	66.1	87.5

DIST

0.0	57.0	23.4	32.2	27.7	40.8	40.2	41.2	8.9
57.0	0.0	57.9	37.5	72.4	78.2	20.6	25.0	51.4
23.4	57.9	0.0	21.6	50.7	21.8	37.4	34.7	16.6
32.2	37.5	21.6	0.0	58.0	40.8	16.9	13.2	23.5
27.7	72.4	50.7	58.0	0.0	64.9	61.0	64.0	36.2
40.8	78.2	21.8	40.8	64.9	0.0	57.6	53.8	36.8
40.2	20.6	37.4	16.9	61.0	57.6	0.0	6.3	33.0
41.2	25.0	34.7	13.2	64.0	53.8	6.3	0.0	33.3
8.9	51.4	16.6	23.5	36.2	36.8	33.0	33.3	0.0

STEP = 30

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
8.9	20.6	16.6	13.2	27.7	21.8	6.3	6.3	8.9
23.4	25.	21.6	16.9	36.2	36.8	16.9	13.2	16.6
27.7	37.5	21.8	21.6	50.7	40.8	20.6	25.0	23.5
32.2	51.4	23.4	23.5	58.0	40.8	33.0	33.3	33.0
40.2	57.0	34.7	32.2	61.0	53.8	37.4	34.7	33.3
40.8	57.9	37.4	37.5	64.0	57.6	40.2	41.2	36.2
41.2	72.4	50.7	40.8	64.9	64.9	57.6	53.8	36.8
57.0	78.2	57.9	58.0	72.4	78.2	61.0	64.0	51.4

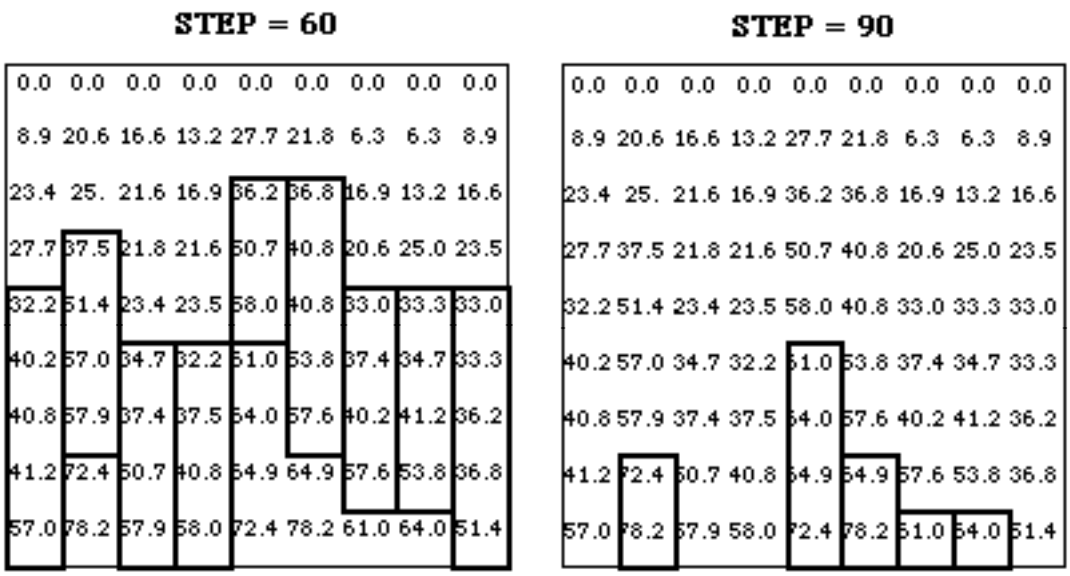

Figure 1. "Sorting" algorithm: using logical masks to reduce calculations in the *computation phase*.

Figure 1 shows the X and Y coordinates for 9 points, their corresponding Z values, the resulting DIST interpoint distance matrix, and how a column-sorted DIST matrix is used to create successive "masks" that are applied to the ZZ matrix. In this example, we assume a maximum radius of 90 units and show the execution for 3 radius increments of 30 units, resulting in STEP values of 30, 60, and 90. For a particular radius value, a highlighted rectangle defines a mask that is applied to the corresponding locations in ZZ. Only those elements of ZZ that fall within the mask are summed for each particular STEP.

This mask is implemented using an array, called **startindex**, that for each radius value, contains the beginning positions of each column. The startindex array for this example is shown in Figure 2.

Initially:

2	2	2	2	2	2	2	2	2

After radius circle d=30:

5	4	6	6	3	3	5	5	5

After radius circle d=60:

10	8	10	10	6	8	9	9	10

After radius circle d=90:

10	10	10	10	10	10	10	10	10

Figure 2. "Sorting" algorithm: the *startindex* array for the sample problem.

3.2 Results: Single Processor

In the first set of analyses, we simply ported the PC version of the "naive" and "sorting" implementations to the KSR-1 and used only one processor. The data set size was increased in 4 steps of 1K, from 1K to 5K observations (1K = 1024 observations) and a large 10K observation data set was also compiled to examine the effect of a very large problem on the performance of the algorithms. The points were derived using a two-dimensional probability surface obtained using the Slatec library, a public domain package of mathematical and statistical functions written in Fortran 77 and provided by Lawrence Livermore National Laboratory. Tables 3 and 4 show a set of timings for these implementations. Table 4 contains an additional row (line 2, labeled "Sort") added to each analysis to show the amount of computation time required to perform the sorting.

Table 3. "Naive" algorithm results (seconds) using a single KSR-1 processor.

	1K	**2K**	**3K**	**4K**	**5K**	**10K**
Initialization	6.56	28.93	65.00	122.24	195.40	1025.1
Computation	26.18	127.26	378.5	668.20	1086.20	4894.58
Total	32.74	156.24	443.50	790.44	1281.6	5919.68

Table 4. "Sorting" algorithm results (seconds) using a single KSR-1 processor.

	1K	**2K**	**3K**	**4K**	**5K**
Initialization	11.00	47.62	115.10	215.78	317.10
Sort	115.76	1373.90	4691.34	11059.0	21683.5
Computation	2.80	13.16	37.06	71.64	119.14
Total	129.56	1434.68	4843.50	11346.4	22119.7

Note that for the “sorting” implementation in Table 4, no entry is given for the 10K data set. We were unable to process this largest data set in the “sorting” experiments because it uses twice as much memory as the “naive” approach and sufficient memory was unavailable; the “sorting” approach uses DIST and ZZ, while only DIST is needed in the “naive” implementation.

Table 3 and Figure 3 show that the performance of the serial “naive” algorithm is non-linear: while the data set size increases by an order of magnitude, the total runtime increases by more than two orders of magnitude.

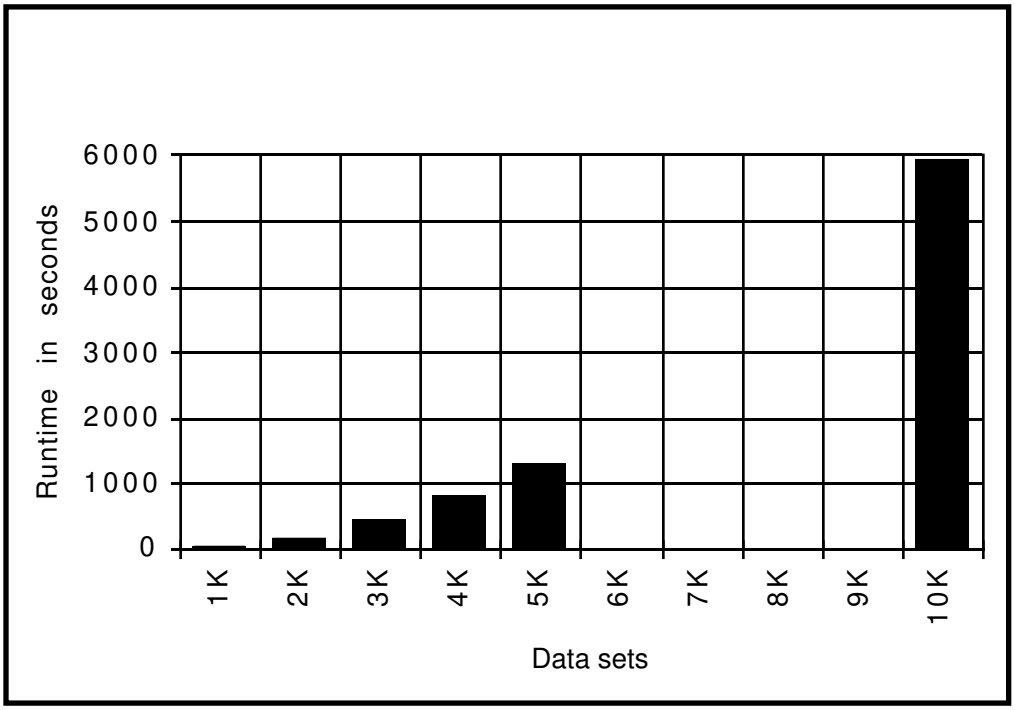

Figure 3. Total runtimes for the serial “Naive” implementation.

Table 4 and Figure 4 show that the overall performance for the “sorting” version also is unsatisfactory, especially when compared to the “naive” version.

On the other hand, if we restrict our attention to the second part of the algorithm (the *computation phase*) the “sorting” version substantially outperforms the “naive” version (Figure 5). Here the potential benefits of restructuring the algorithm are apparent. Each column of the distance matrix is traversed only once for all radius values (all STEP values in the code), as opposed to the "naive" implementation, where columns are scanned in their entirety for each radius value.

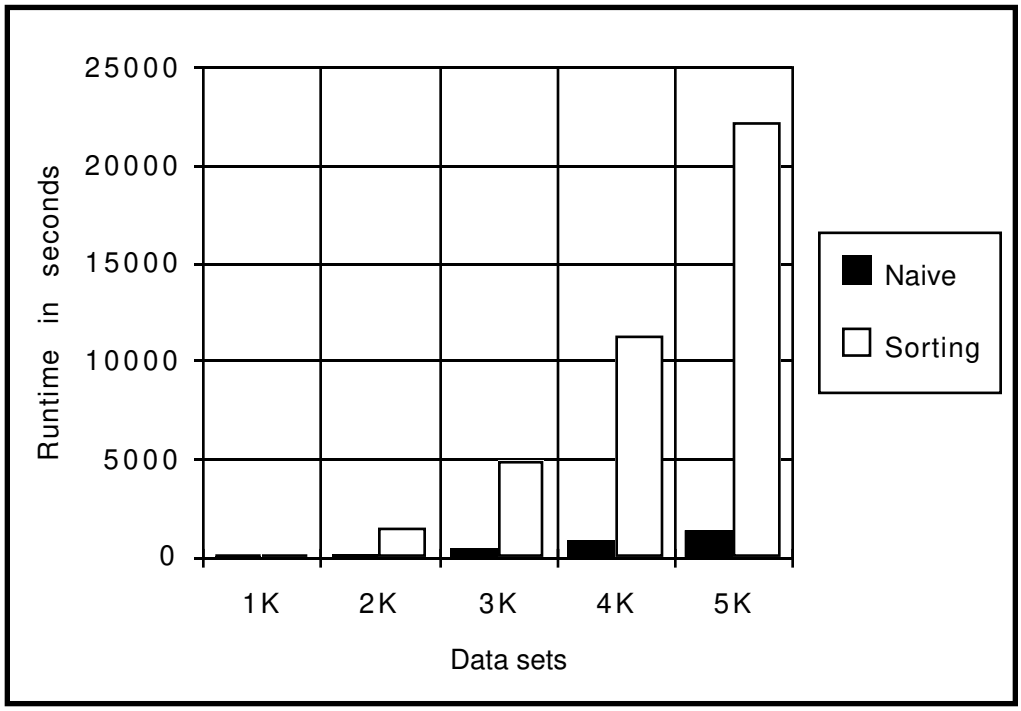

Figure 4. Total runtimes for the serial "Naive" and "Sorting" implementations.

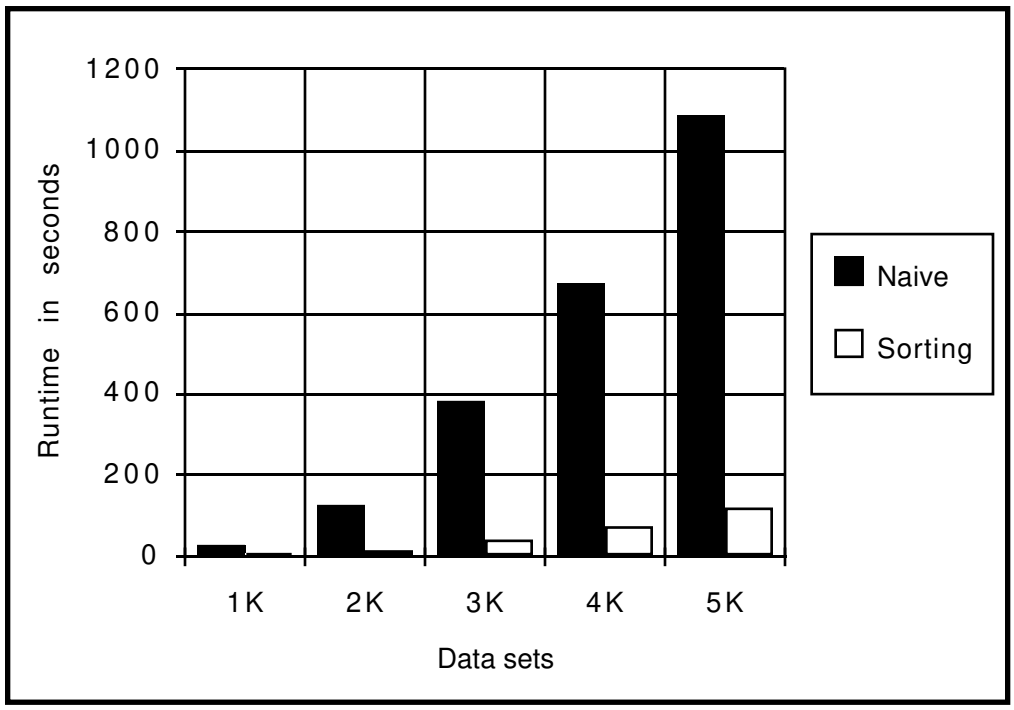

Figure 5. *Computation times* (only) for the serial "Naive" and "Sorting" implementations.

In the next section, we evaluate the performance of the statistical analyses using the two approaches and illustrate how they were adapted to the parallel KSR-1 environment.

4.0 PARALLEL IMPLEMENTATIONS

In the previous section, we demonstrated that the "sorting" implementation performed better than the "naive" version in the *computation phase*. However, the "naive" version still performed better overall because of the overhead of sorting the distance matrix.

4.1 Parallel Algorithm Modifications

Parallelizing the *sorting phase* is crucial since this is the most time-consuming step and if we can reduce sorting times, we are guaranteed to improve execution in the *computation phase*.

The skeletal structure of the original sorting code appeared as:

```
        DO 66 J=1, NOBS
                DO 65 I=1, NOBS
c                       sorting code
 65             continue
 66     continue
```

In the parallel version we inserted a tiling directive:

```
c*ksr* USER TILE  ( J, STRATEGY=slice, PRIVATE=(I) )
        DO 66 J=1, NOBS
                DO 65 I=1, NOBS
c                       sorting code
 65             continue
 66     continue
c*ksr* END TILE
```

This simple change ensured that the NOBS iterations in the J direction are split into as many groups as there are available cells. Each cell is responsible for sorting the J indices it has been assigned. Thus the sorting of the distance matrix is done in parallel by column. Six additional modifications were applied to both versions of the algorithm to improve their performance in the parallel KSR-1 environment.

(1) Loops were reordered to increase locality of reference.

Because of the hierarchical cache structure of the KSR-1 (see Table 2), loop ordering can have a dramatic effect on runtimes. By inverting the subscripts of the nested loop used to compute distances, we increased performance by reducing the number of memory accesses with high latencies.

```
        DO 2 I=1,NOBS
                DO 2 J=1,NOBS
                        IF (I .NE. J) THEN
                                XDIST = X(I) - X(J)
                                YDIST = Y(I) - Y(J)
                                DIST(I,J) = SQRT((XDIST*XDIST) + (YDIST*YDIST))
                        ENDIF
2       CONTINUE
```

became:

```
        DO 2 J=1,NOBS
                DO 2 I=1,NOBS
...
C  * lines deleted *
...
2       CONTINUE
```

This simple optimization reduced the time required to initialize the DIST matrix (for the serial version and for NOBS=5120) from 10,000 seconds to 195 seconds. By making changes such as this we avoid needless communication among caches ("thrashing"), and associated latency penalties.

(2) Data were aligned on subpage boundaries

A subpage is the smallest unit of data to be manipulated as a unit. It is 128 bytes long, or 16 64-bit words. To improve performance on the KSR-1, the leading index of each array must be a multiple of 16. We used the:

```
c*ksr*subpage X,Y,Z,DIST
```

compiler directive to align X,Y,Z,and DIST to subpages and thereby minimize data movement and the possibility of thrashing when writing data.

(3) A team of pthreads was explicitly allocated by calling the following function:

```
call ipr_create_team( nthreads, idteam )
```

When a thread encounters a begin-parallel directive (such as **c*ksr* tile**), it summons a team of pthreads. If such a team is not explicitly defined by the programmer, the runtime library (called Presto) creates a team and assigns it to that parallel region. When a team reaches the end-parallel directive (such as **c*ksr* end tile**), it is usually disbanded and its members are returned to the idle pool. Creation and disbanding of a team do not imply creation and destruction of pthreads, as only the grouping of pthreads and the assignment of team identification numbers occur on these team operations. Since the runtime library creates and disbands a team for each parallel region, thus creating extra overhead, it is more efficient to create a team of pthreads at the start of a program and reuse that team in each parallel region of code.

(4) Semi-automatic tiling was used

There are three approaches to develop a parallel program from a serial version: **(1) automatic approach**, where the programmer writes a sequential FORTRAN 77 program and invokes the compiler to automatically transform the code into a parallel version (the compiler "does everything"); **(2) manual approach**, where it is the programmer's responsibility to modify the sequential program by inserting parallel constructs such as parallel regions, parallel sections, tile families, and affinity regions (the programmer "does everything"); **(3) semi-automatic approach**, where the programmer modifies the sequential program by assigning indices and efficiency-related parameters (e.g. tile directives) and then invokes the compiler to perform the remaining modifications and optimizations (the programmer and the compiler "work together"). We used the semi-automatic approach. This has advantages over the automatic approach, which is based on syntax only, because the programmer's knowledge of the problem size and the effects of non-local cache access on runtime efficiency can be used to select the best tiling approaches. Our code (*initialization* and *computation*) with semi-automatic tiling and before the use of KAP (see (5)) became:

```
c*ksr* ptile (i,j,TEAMID=idteam,STRATEGY=slice)
        DO 2 I=1,NOBS
                DO 2 J=1,NOBS
C * code deleted *
2       CONTINUE

        ...

c*ksr* ptile (j,TEAMID=idteam,STRATEGY=slice)
        S2 = 0.
        ZSUM = 0.
        ZS = 0.
        WT = 0.
        DO 3 J=1,NOBS
                WA = 0
                DO 3 I=1,NOBS
C * code deleted *
        3       CONTINUE

        G = ZS / ZSUM

c*ksr* ptile (j,TEAMID=idteam,STRATEGY=slice)
        DO 4 J=1,NOBS
                S2 = S2 + W(J)*4*W(J)
4       CONTINUE
```

(5) KAP was run on the source file that was "semi-automatically modified"

After the programmer has modified the source code using semi-automatic parallel directives, a compiler tool called KAP is used on this code. The result of running KAP is a parallel code that has benefited from the programmer's insights as well as from the compiler's abilities. KAP is a source-to-source preprocessor that performs code optimizations and parallelizes loop nests by inserting explicit tile directives to represent the parallelism. KAP also recognizes and handles requirements for private and reduction variables. Private variables are used to ensure that parallel calculations are performed on a specific, local subset of the data; reduction variables are used to accumulate values to be used when summary calculations are made by a single, master processor. By running KAP on the previously semi-automatic tiled file, the following file was automatically generated:

```
c*ksr* TILE (i,j,PRIVATE=(XDIST,YDIST),TEAMID=idteam,
c*ksr*&                    STRATEGY=slice)
        DO 2 I=1,NOBS
                DO 2 J=1,NOBS
C * code deleted *
 2      CONTINUE
c*ksr* END TILE

        ...

c*ksr* TILE (j,PRIVATE=(WA,I),TEAMID=idteam,
c*ksr*&                    REDUCTION=(ZSUM,ZS,WT),
c*ksr*&                    STRATEGY=slice)
        S2 = 0.
        ZSUM = 0.
        ZS = 0.
        WT = 0.
        DO 3 J=1,NOBS
                WA = 0
                DO 3 I=1,NOBS
C * code deleted *
3       CONTINUE
c*ksr* END TILE

        G = ZS / ZSUM

c*ksr* TILE (j,TEAMID=idteam,REDUCTION=(S2),
c*ksr*&                    STRATEGY=slice)
        DO 4 J=1,NOBS
                S2 = S2 + W(J)*4*W(J)
4       CONTINUE
c*ksr* END TILE
```

(6) AFFINITY REGION directives were manually inserted

Affinity regions are a parallel construct used to improve data locality. An affinity region would typically be used when consecutive tile families must access one or more of the same arrays. Using tile families inside an affinity region causes the same pthread-to-tile assignments. In other words, a ptile accessing part of an array in the first tile family would be guaranteed to access that same array section in the second tile family. Without this mechanism, a ptile in the second family might be allocated a different section of the array and this could result in a reloading of its local cache. Thus affinity regions ensure that data movement is minimized and cache performance is maximized. The following example illustrates the "embedding" of two tile families into one affinity region that specifies which loop indices are *common* to both tile families:

```
c*ksr*  AFFINITY REGION (J:1,NOBS, TEAMID=idteam, STRATEGY=slice)

c*ksr* TILE (J, PRIVATE=(WA,I), REDUCTION=(ZSUM,ZS,WT))
        S2 = 0.
        ZSUM = 0.
        ZS = 0.
        WT = 0.
        DO 3 J=1,NOBS
                WA = 0
                DO 3 I=1,NOBS
C * code deleted *
3       CONTINUE
c*ksr* END TILE

        G = ZS / ZSUM

c*ksr* TILE (J, REDUCTION=(S2))
        DO 4 J=1,NOBS
                S2 = S2 + W(J)*4*W(J)
4       CONTINUE
c*ksr* END TILE

c*ksr* END AFFINITY REGION
```

4.2 Parallel "Naive" Results

We conducted the parallel experiments using 1, 7, 16, 29, and 59 cells because at the time the analyses were conducted, these were the only configurations available. The number of pthreads used in a particular run is specified by assigning a value to an environmental variable called PL_NUM_THREADS (i.e., "setenv PL_NUM_THREADS 16", will set the pool of available pthreads to 16). The number of processors or cells allocate to run those pthreads is specified at runtime either interactively or through a batch queue assignment. An *interactive* example would be *"allocate_pset s8.1 a.out < in"*. In this command *s8.1* refers to a predefined set of 8 processors on which the executable file *a.out* will be run, using input parameters redirected from a file called *in*. A batch example would be *"qsub -q large jobfile"*, where the *qsub* command is used to submit a batch request to the large queue consisting of 64 processors.

The results displayed in Table 5 show that substantial performance improvements can be gained using parallel processing. The times obtained for the single processor runs clearly increase as a function of problem size. While 32.74 seconds are required to analyze a 1K point dataset, when the number of points is increased by an order of magnitude to 10K, the runtime increases to 5919.68 seconds (99

minutes), an increase of two orders of magnitude. This turnaround time for the 10K dataset is unacceptable for interactive analyses. With 59 cells, however, total runtime is reduced to 125 seconds.

Table 5. "Naive" algorithm runtimes (seconds) for different datasets and numbers of cells.

		1	**7**	**16**	**29**	**59**
1 K	Initialization	6.56	2.50	2.30	1.44	2.04
	Computation	26.18	3.18	1.62	1.30	1.00
	Total	**32.74**	**6.08**	**3.92**	**2.74**	**3.04**
2 K	Initialization	28.93	6.30	4.26	3.86	4.36
	Computation	127.26	12.16	6.12	3.88	2.48
	Total	**156.24**	**18.46**	**10.38**	**7.74**	**6.84**
3 K	Initialization	65.00	12.90	8.24	7.58	8.46
	Computation	378.5	27.11	13.62	8.04	4.8
	Total	**443.50**	**40.01**	**21.86**	**15.62**	**13.26**
4 K	Initialization	122.24	18.04	13.96	11.50	12.38
	Computation	668.20	48.85	24.06	13.72	7.82
	Total	**790.44**	**66.89**	**38.02**	**25.22**	**20.20**
5 K	Initialization	195.40	25.90	17.48	15.60	16.04
	Computation	1086.20	78.83	38.42	23.34	11.96
	Total	**1281.6**	**104.73**	**55.9**	**38.94**	**28.00**
10 K	Initialization	1025.1	225.76	171.28	121.08	80.46
	Computation	4894.58	504.67	240.17	118.39	44.28
	Total	**5919.68**	**730.43**	**411.45**	**239.47**	**124.74**

For small problem sizes, the overhead of establishing parallel processes may degrade performance; this is evident in the execution times for the 1K problem. Since the fixed cost of process creation comprises a significant proportion of the processing time, this leads to an actual increase in runtime as the number of processors is increased from 29 to 59. The other, larger datasets (2K to 10K) scale nicely with the number of cells and reach their maximum performance, as measured by total time, when the maximum (59) number of cells is used. This is demonstrated best in the bottom line of Table 5, which shows the effect of adding processors to the computation of results for the largest problem (10K). Total times decrease from more than one and a half hours to

approximately two minutes as the number of processors increases from 1 to 59. Thus, for the 10K point dataset, the total runtime obtained for 59 cells makes the problem suitable for interactive analysis. Figure 6 shows this dramatic decrease of total runtime when using 7, 16 , 29 or 59 processors.

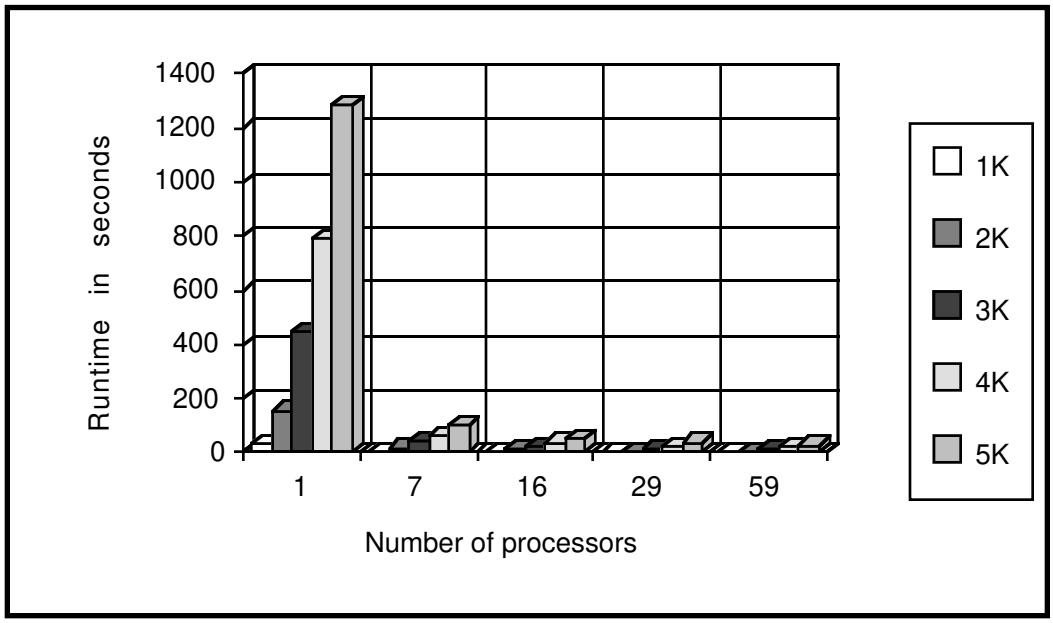

Figure 6. "Naive" algorithm: parallel runtimes for the 1K, 2K, 3K, 4K, and 5K data sets.

Speedup is a measure that is often used to assess the improvement of a parallel implementation relative to the performance obtained using a single processor:

$$\text{Speedup} = \frac{Execution_Time_for_1_cell}{Execution_Time_for_N_cells}$$

The results, standardized as a ratio of the two times, indicate that the set of available processors is being used efficiently in many of the experiments. Though the maximum speedup normally obtainable is equal to the number of processes (Brawer, 1989: 75), the results shown in Figure 7 indicate that in some instances *superlinear* speedups were obtained in our experiments. This superlinearity was investigated further by dividing the problem into the two sections used earlier to examine the computational requirements of the problem: *initialization* and *computation*.

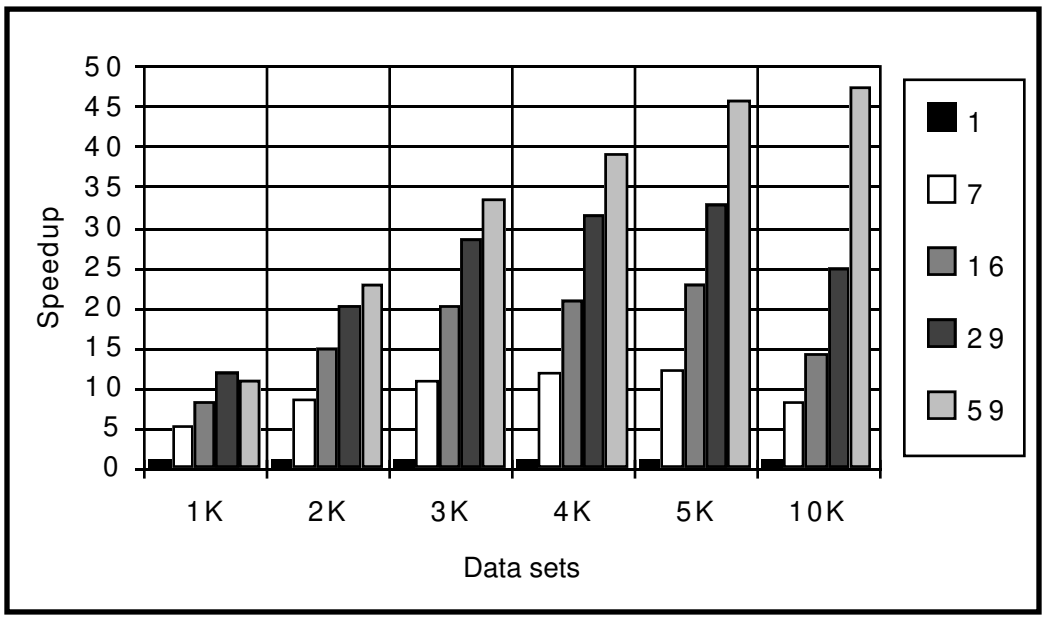

Figure 7. "Naive" algorithm: Total runtime speedups for the 1K, 2K, 3K, 4K, 5K and 10K data sets.

Figure 8 focuses on *initialization* speedup and shows that when 7, 16 or 29 cells are used, speedup increases across data sets of sizes 1K to 5K, but then decreases for the largest (10K) data set. When all 59 cells were used, however, the speedup increases monotonically across all six data sets.

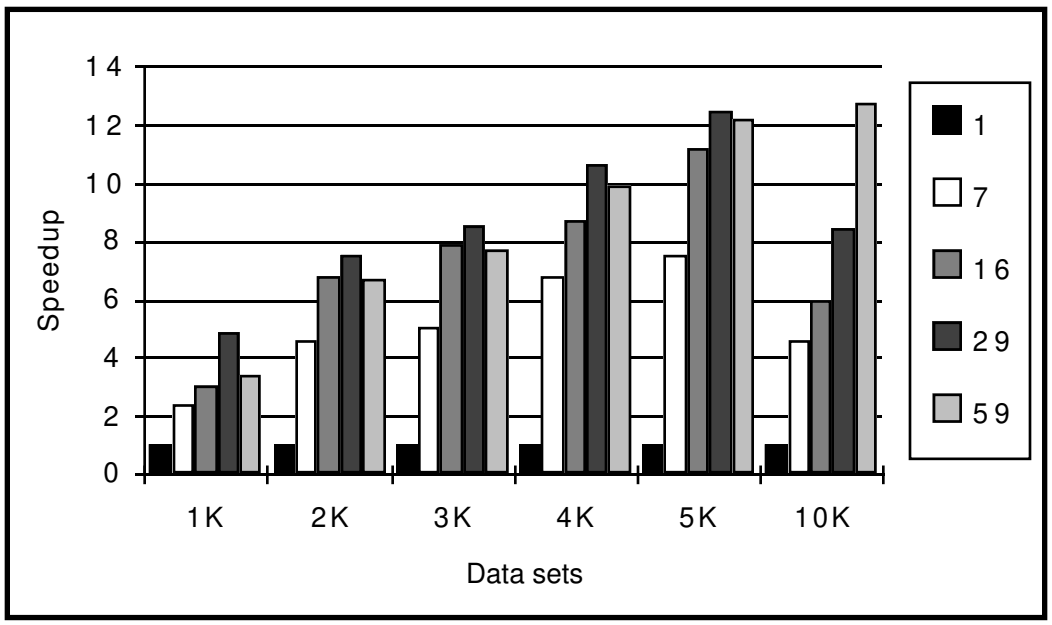

Figure 8. "Naive" algorithm: *Initialization time* speedups for the 1K, 2K, 3K, 4K, 5K and 10K data sets.

Figure 9 shows *computation time* speedups. Again note that only when 59 cells were used was there a monotonic increase in speedup across all data sets. When only 29 cells were used, speedup starts to drop after the 4K data set. Also note in Figure 9 that the "superlinearity" of speedups is quite evident.

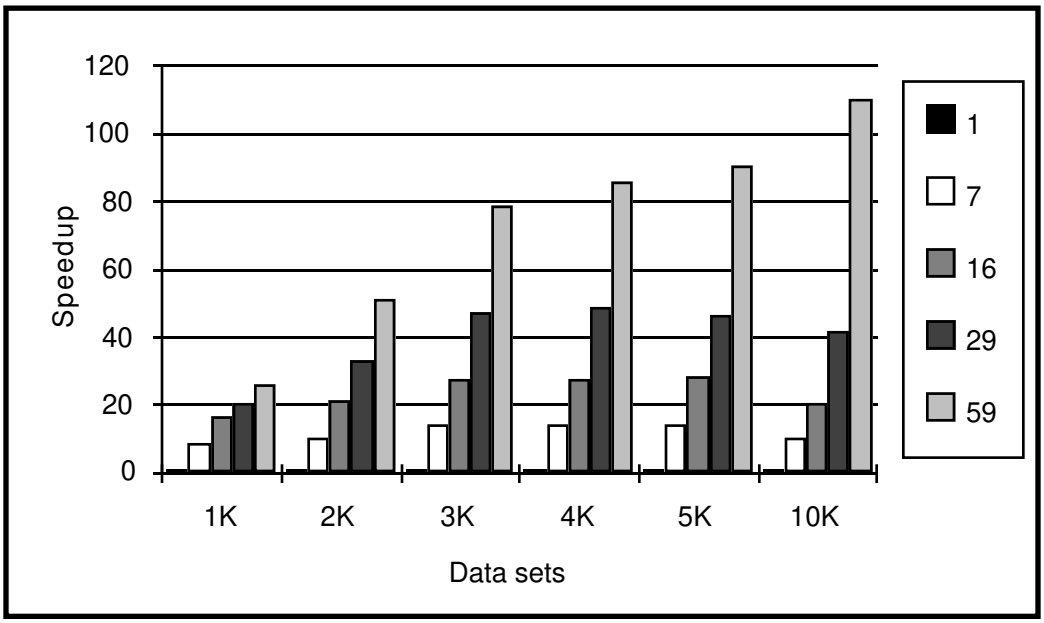

Figure 9. "Naive" algorithm: *Computation time* speedups for the 1K, 2K, 3K, 4K, 5K and 10K data sets.

Superlinear Speedups:

Superlinear speedups occur when observed speedup values are greater than the number of processors used. In our analyses this occurred, for example, when we used 59 processors to analyze the 3K, 4K, 5K, and 10K data sets. In those cases we obtained speedups greater than 70 and, in one case, the speedup actually exceeded 110 on a 59 processor system. The explanation for this "abnormal" speedup lies in the hierarchical architecture of the computer and how tasks are allocated to cells to exploit the structure of the problem. If, for example, a large memory segment, one that exceeds the capacity of N processors (N times the capacity of 1 processor's 32 Mb local cache), can be divided equally into M parts (M > N) where the size of each of these parts is less than 32 Mb, each of the M processors is able to access all of its memory within its local cache and never requires access memory on the hierarchical ring structure. Consequently, it does not have to flush its local cache and incur latency penalties (e.g. 175 clock cycles) for non-local memory access (see Table 2).

Table 6 provides a rough estimate of a lower bound on the size of the data sets that would cause non-local memory accesses to occur. Since DIST is the only 2-dimensional matrix and is the largest data structure in the "naive" implementation, it is used to characterize the problem shown in Table 6: the first column represents the data set size, the second the number of elements in the DIST matrix (this serves as a lower bound on the total size of the data structures required), the third the space in megabytes occupied by DIST (on the KSR-1, a floating point variable occupies 8 bytes or 64 bits), the fourth the minimum number of cell caches required to store DIST (a cell's cache capacity is 32MBytes), and the fifth and last column the minimum number of processors needed (1, 7, 16, 29, or 59).

Table 6. "Naive" algorithm: minimum number of cells needed for each dataset in order to obtain locality of data.

N elements	elements in DIST(N,N)	Space in MBytes	minimum #32MB caches	minimum #cells needed
1 K	1 Meg	8	1	**1**
2 K	4 Meg	32	2	**7**
3 K	9 Meg	72	3	**7**
4 K	16 Meg	128	5	**7**
5 K	25 Meg	200	7	**7**
6K	36 Meg	288	10	**16**
7K	49 Meg	392	13	**16**
8K	64 Meg	512	17	**29**
9K	81 Meg	648	21	**29**
10 K	100 Meg	800	26	**29**

If we assume that the data are partitioned evenly and allocated to the different cells, as the dataset size reaches 2K elements, the amount of storage available on a single local cache is exceeded. When using 7 or more cells, problem sizes ranging from 2K to 5K points can fit into local cache (Figure 10).

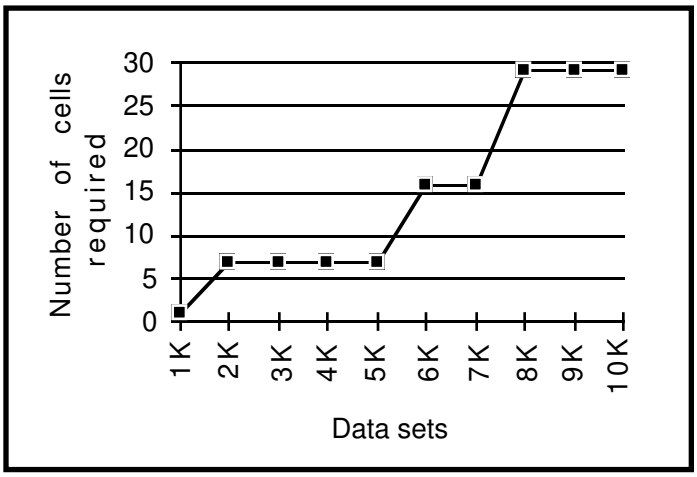

Figure 10. "Naive" algorithm: number of cells vs. dataset size needed to obtain locality of data.

When dealing with very large problem sizes, such as the 10K data set, however, at least 29 cells must be used to access all data locally. As a consequence of this stepped response, speedups are most meaningful only when comparing datasets that have similar space requirements. In other cases non-local memory references can cause abrupt discontinuities in performance.

4.3 Parallel "Sorting" Results

The "sorting" implementation of the **G**(d) program was run using the same set of processors and datasets as reported in Table 5. These results are shown in Table 7, with an additional row (line 2, labeled "Sort") added to each analysis to show the amount of computation time required to perform sorting.

For a given data set size (e.g., 1K, 2K), we can see how sorting times (e.g., line 2 in Table 7), generally decrease as the number of processors increases. This shows that our objectives of reducing the sorting phase's runtime are met. The total runtime entries (line 4, labeled "Total") are, however, greater than their parallel "naive" counterparts (Table 5).

Table 7. "Sorting" algorithm runtimes (seconds) for different datasets and numbers of cells.

		1	**7**	**16**	**29**	**59**
1 K	Initialization	11.00	5.64	3.90	5.58	18.56
	Sort	115.76	38.70	7.82	5.92	5.12
	Computation	2.80	0.74	0.18	0.16	0.36
	Total	129.56	45.08	11.90	11.66	24.04
2 K	Initialization	47.62	21.66	16.08	27.58	50.72
	Sort	1373.90	298.26	62.64	39.22	23.62
	Computation	13.16	1.38	0.34	0.36	0.28
	Total	1434.68	321.3	79.06	67.16	74.62
3 K	Initialization	115.10	56.54	34.60	69.14	133.84
	Sort	4691.34	990.28	212.36	124.08	71.02
	Computation	37.06	20.06	0.68	0.44	0.36
	Total	4843.50	1066.88	247.64	193.66	205.22
4 K	Initialization	215.78	82.44	69.28	136.20	260.58
	Sort	11059.0	1169.84	503.28	283.84	158.28
	Computation	71.64	5.74	1.24	0.66	0.54
	Total	11346.4	1258.02	573.8	420.7	419.4
5 K	Initialization	317.10	110.14	118.20	202.52	418.10
	Sort	21683.5	2272.18	982.18	588.74	294.74
	Computation	119.14	10.26	2.60	1.28	0.74
	Total	22119.7	2392.58	1102.98	792.54	713.58

4.4 Comparison of Parallel "Sorting" with Serial "Naive"

When the parallel "sorting" version results are compared to the serial "naive" implementation, two large differences in performance can be seen. First, as shown in Figure 11, the total runtimes for the parallel "sorting" version with 59 cells are substantially better than the serial "naive" version of the code. This demonstrates the benefits of parallelism.

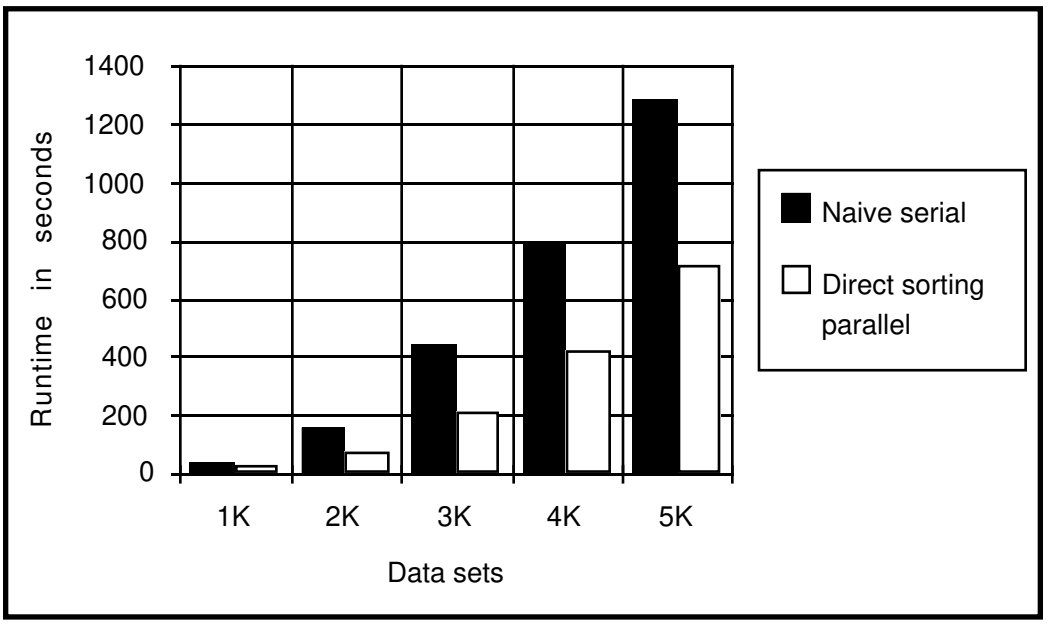

Figure 11. Comparing the Total runtimes for the serial "Naive" and parallel "Sorting" with 59 cells.

Second, as shown in Table 7, the *computation times* are much lower in the parallel "sorting" version. This is well illustrated in Figure 12.

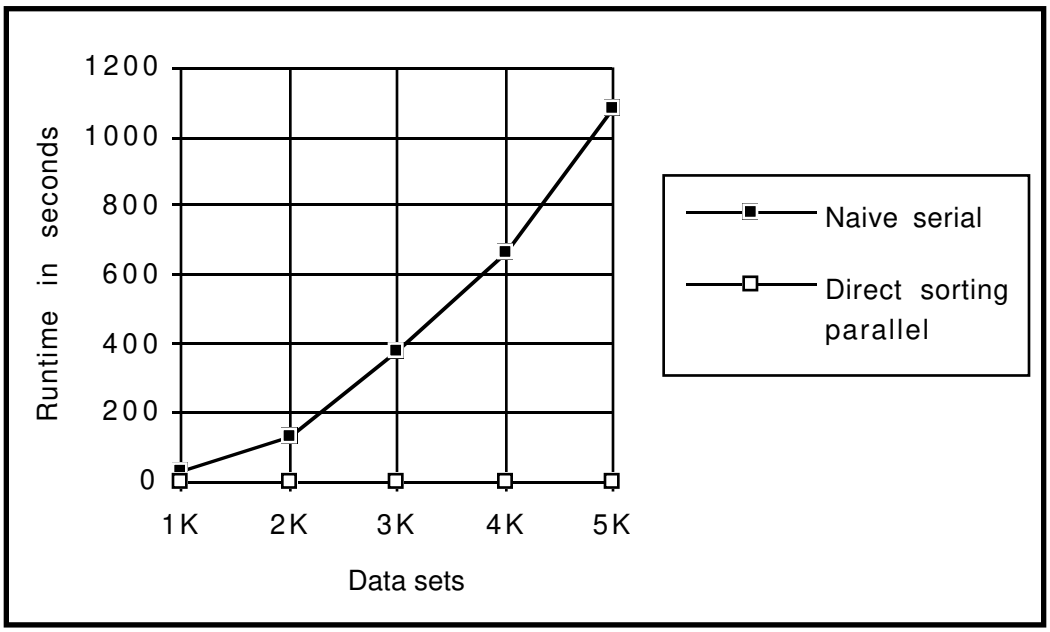

Figure 12. Comparing the *Computation runtimes* for the serial "Naive" and parallel "Sorting" with 59 cells.

The speedups that compare the *computation step* performance of these two versions for different problem sizes ranging from 1K to 5K (Table 8), increase from 75 for the 1K point problem, to 1468 for the 5K point problem.

Table 8. *Computation phase* speedup (parallel "Sorting" over serial "Naive").

	1K	2K	3K	4K	5K
Speedup	75	455	1051	1237	1468

Also note that in the serial "naive" implementation *computation time* was about 4 times slower than the *initialization time*. In the "sorting" approach with parallelism, the bottleneck becomes *initialization time* (distance matrix sorting time), not *computation time*.

4.5 Comparison of the Parallel Implementations

Figure 13 compares total runtimes for the two parallel implementations. In the 5K point analysis, the parallel "sorting" version takes almost 12 minutes while the parallel "naive" version takes about 28 seconds.

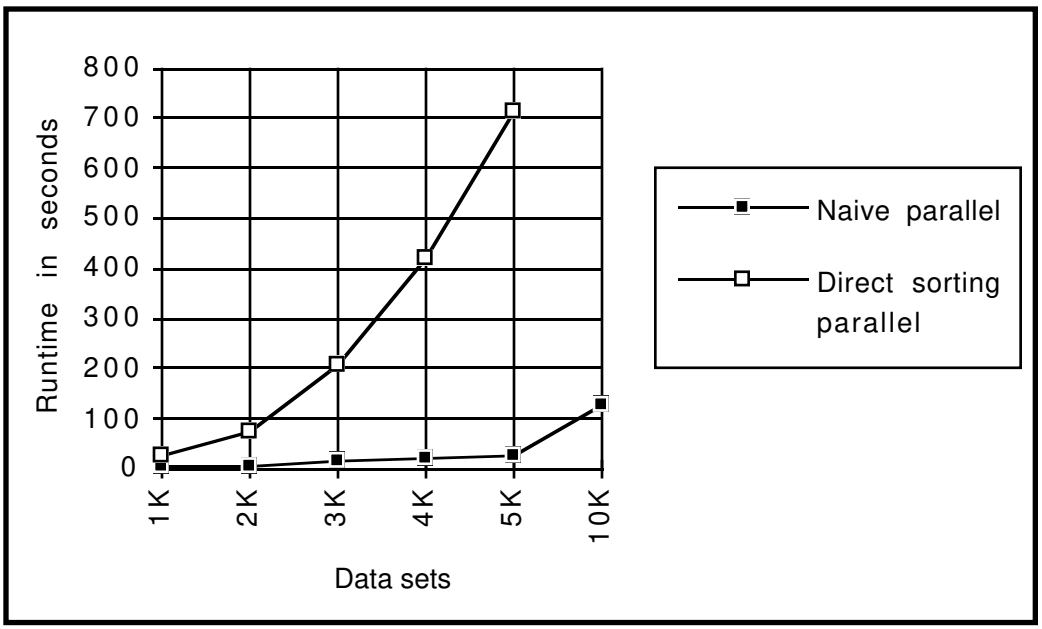

Figure 13. Comparing the Total runtimes for the parallel "Naive" and parallel "Sorting" versions (59 cells).

However, if we examine times obtained for the *computation step* only, then the performance advantage of sorting again becomes clear (Figure 14): almost 12 seconds for the parallel "naive" version, while the parallel "sorting" version takes 0.74 seconds.

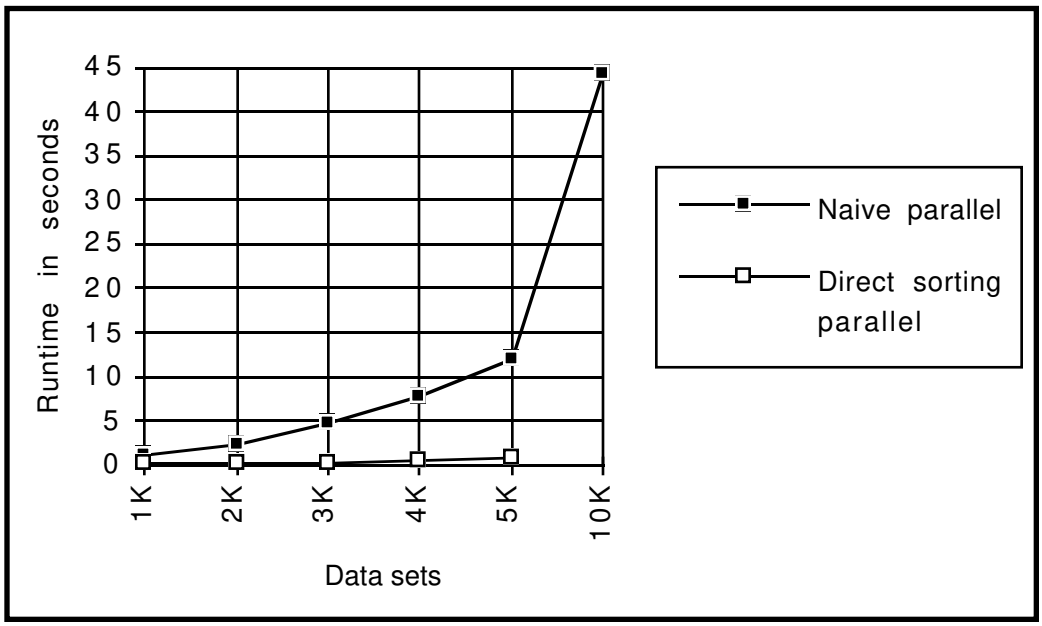

Figure 14. Comparing the *Computation runtimes* for the parallel "Naive" and parallel "Sorting" versions (59 cells).

The choice of a particular method to use depends on the size of the data set, and the number of iterations (*d*-steps) in the *computation phase*. Since sorting adds additional overhead but is only performed once, it is best suited for problems in which the number of iterations is very large (in our timing example, we only used 20 *d*-steps) and when multiple runs that use different step sizes are contemplated. In such cases, subsequent statistical calculations can be performed efficiently on the sorted distance structures without incurring the sorting overhead penalty.

5.0 CONCLUSIONS

In this paper we demonstrated that a conventional serial implementation of a measure of spatial association exhibits a geometrical explosion in response time as

increasingly large datasets are analyzed. A parallel supercomputer was used to overcome these limitations. Our experiments compared a straightforward recasting of the serial code into a parallel version with a new version that was intended to exploit the structure of the problem in the parallel environment. Both parallel versions clearly outperformed their serial counterparts, though the overhead incurred by sorting reduced the overall performance obtained in that set of experiments. Nevertheless, total runtimes were substantially reduced across the entire range of problem sizes analyzed. In most cases, but especially when large problems were run, the full complement of processors was used efficiently, as near-linear, and in some cases superlinear, increases in speedup were obtained in our implementations.

Further work should be done in three areas. First, the results we reported for the sorting implementation uses a sorting technique that is not especially known for its speed, since we focused on ease of programming instead. This implementation is based on a fairly efficient *shellsort* algorithm given in Sedgewick (1990). By moving to faster sorts, such as the *quicksort*, we might further improve the sorting runtimes, thus bringing down the total *initialization* overhead. Second, the application of different problem decomposition strategies to the algorithm and the use of alternative data structures may yield improvements in performance. Finally, it is possible that a different parallel computer architecture, such as SIMD, could perform calculations of the type required in our analyses even faster than our implementations on this hybrid machine.

References

Anselin, L. (1991) *Spacestat*. Software available from the National Center for Geographic Information and Analysis, University of California, Santa Barbara, CA.

Anselin L. and Griffith, D. (1988) Do spatial effects really matter in regression analysis? *Papers of the Regional Science Association*. **65**, 11-34.

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

Brawer, S. (1989) *Introduction to Parallel Programming*, Academic Press, San Diego, CA.

Cliff, A.D. and Ord, J.K. (1973) *Spatial Autocorrelation*, Methuen, New York, NY.

Cliff, A.D. and Ord, J.K. (1981) *Spatial Processes: Models and Applications*, Pion, London, UK.

Cressie, N. (1991) *Statistics for Spatial Data*, John Wiley, New York, NY.

Getis, A. and Ord, J.K. (1992) The analysis of spatial association by use of distance statistics. *Geographical Analysis*. **24**, 189-206.

Goodchild, M.F. (1986) *Spatial Autocorrelation*, CATMOG #47. Geo Books: Norwich, UK.

Gould, P.R. (1970) Is Statistix Inferens the geographical name for a wild goose? *Economic Geography*, **46**, 439-448.

Griffith, D. (1990) Supercomputing and spatial statistics: a reconnaissance. *The Professional Geographer*, **42**, 481-492.

Griffith, D. A. (1987) *Spatial Autocorrelation: A Primer*, Association of American Geographers, Washington, DC.

Kendall Square Research. (1992a) *KSR Fortran,* Kendall Square Research Corporation, Waltham, MA.

Kendall Square Research. (1992b) *KSR Parallel Programming*, Kendall Square Research Corporation, Waltham, MA.

Odland, J. (1988) *Spatial Autocorrelation*, Sage Publications, Newbury Park, CA.

Oliver, M., Webster, R. and Gerrard, J. (1989) Geostatistics in physical geography. Part I: theory. *Transactions of the Institute of British Geographers*, **14**, 259-269.

Sedgewick, R. (1990) *Algorithms in C*, Addison-Wesley, New York, NY.