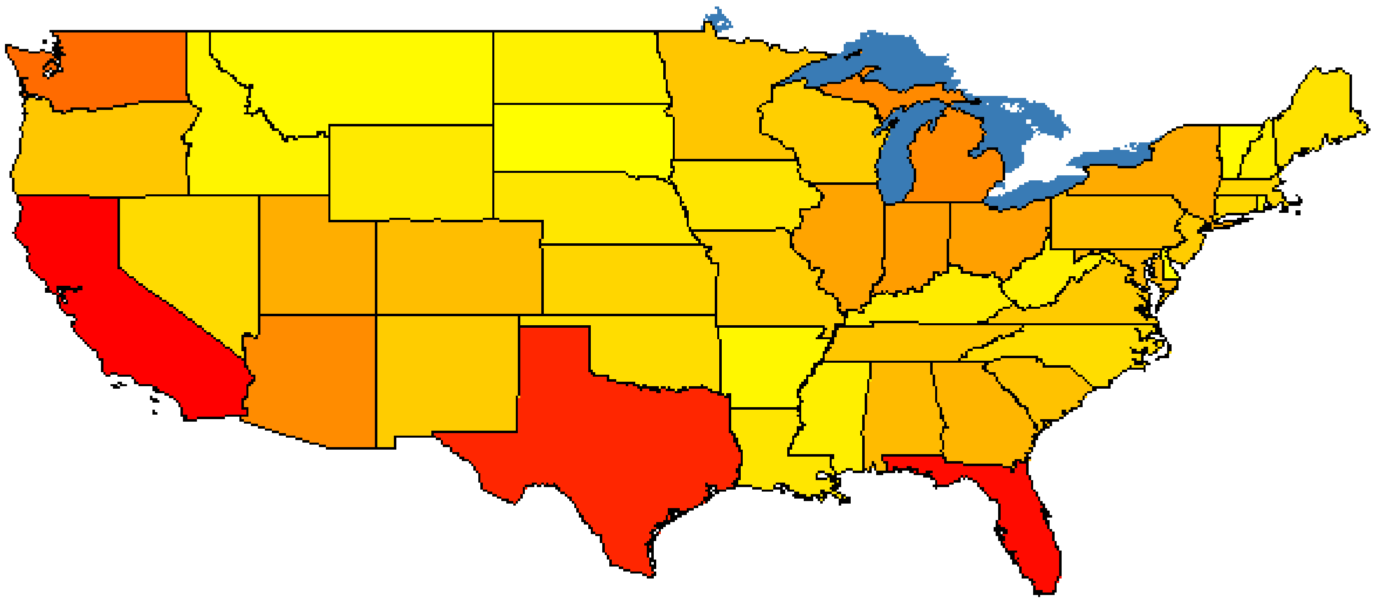


Parallelization of GIS / Computational Geometry Problems

James Jefferson Jarvis jjj@iastate.edu

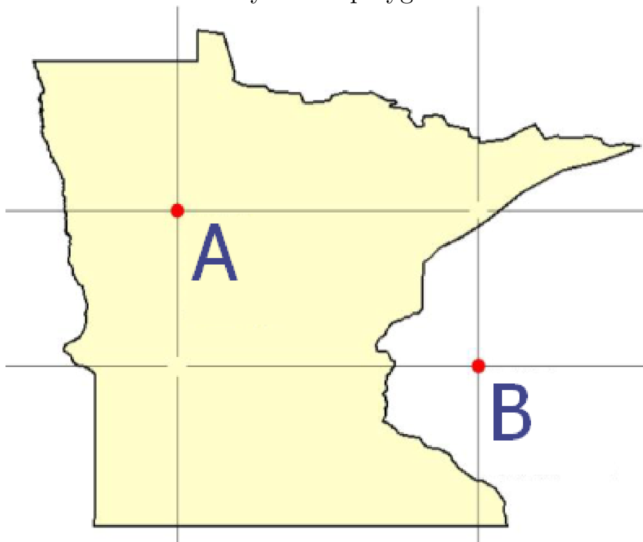
May 5, 2004

0.1 Introduction

I deal with many geographic points (latitude, longitude) and have written programs for finding closest geographic point, nearest road, and polygon that contains the point. My application involves real-time and bulk processing of points. On single processor systems these programs take a long time to run with large number of points and/or large geographic datasets.

I would like to use parallel computing to speed up these operations. One of my applications is to find the nearest road to a geographic point in real-time for about 50 points per second. This is not possible with a single processor workstation but should be possible with parallel processing or high performance computing.

In 1999 I wrote a program called shpdump-poly that can very rapidly determine what polygon a point lies within. This test is performed by drawing an imaginary horizontal and vertical line through the test point. By counting the number of times the line crosses the polygon it is possible to determine if the point falls within or outside the polygon. If, in each of the four cardinal directions, the line crosses the polygon an odd number of times then the point is within the polygon. In order to optimize processing time I load each vertex of the polygon and the associated bounding box into memory. Because bounding box checking is more efficient then point in polygon testing, each test point is first checked to see if it falls within the bounding box. If the point falls within the bounding box it is passed to the slower point in polygon test to determine if it is really in the polygon.



Point A is inside polygon because it has an odd number of edge crossings.

Point B is outside the polygon because there is an even number of edge crossings

The point in polygon algorithm is implemented in C as:

```
int pnpoly(int npol, int start, double *xp, double *yp, double x, double y) {
   int i, j, c = 0;
   for (i = start, j = npol-1; i < npol; j = i++) {
      if ( ( ((yp[i]<=y) && (y<yp[j])) ||
           ((yp[j]<=y) && (y<yp[i]))) &&
           (x < (xp[j] - xp[i])*(y - yp[i]) / (yp[j] - yp[i]) + xp[i])) {
              c = !c;
      }
   }
   return c;
}
```

The serial program has two inputs and one output. It takes a one or more search points and a special datafile that represents the polygons that it searches. The search point consists of double precision values for latitude and longitude and an 11 character string for a label of the point. The polygon geometry is read from file in Shapefile format. The shapefile format is an industry standard way of representing GIS data that consists of points, lines, and polygons.

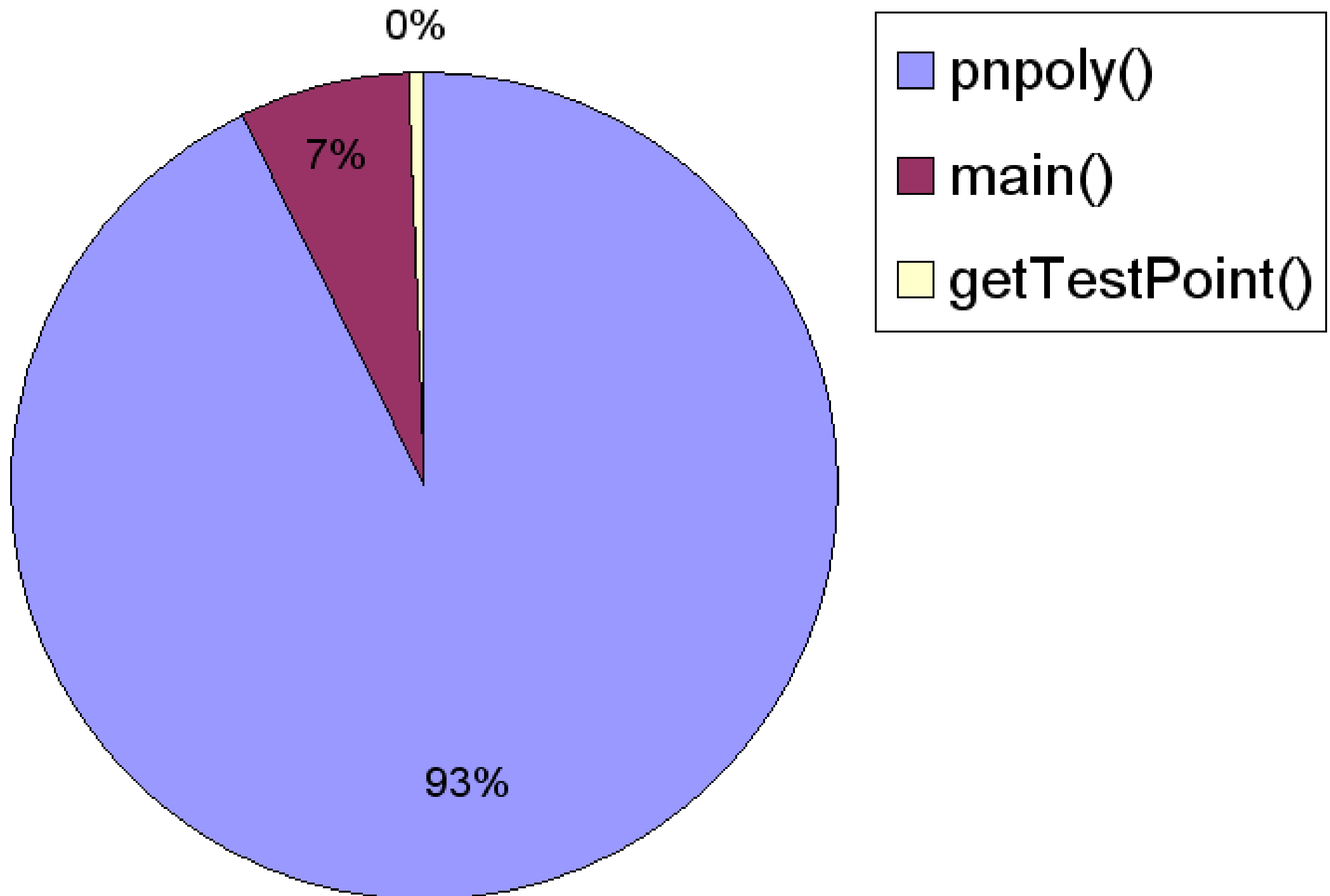
The search points are read from ASCII text file with the three values separated by white space (typically tabs) and records delimited with newlines. This input is handled by C's standard fgets() and fscanf() functions. The following is an

example of input data:

```
46.784585   -90.707886  KB0THN
44.265333   -78.372500  SSF
38.739667   -83.839667  KC8OVO
35.092167   -80.925667  WA4SAS-14
39.418000   -92.439167  KC0RCW
```

The shapefiles are handled by linking to a library called Shapelib (http://shapelib.maptools.org). By using the shapelib API it is possible to open the binary shapefiles and represent the polygons by arrays of double precision numbers. Shapefiles also allow attributes to be specified for shapes (example: the name and population of a state) and Shapelib handles this.

By profiling the original shpdump-poly program I determined that the program usually spent 92% percent of its CPU time in the polygon searching function. Because the polygon searching function is a simple loop with many floating point comparisons I determined the program's performance was CPU bound.

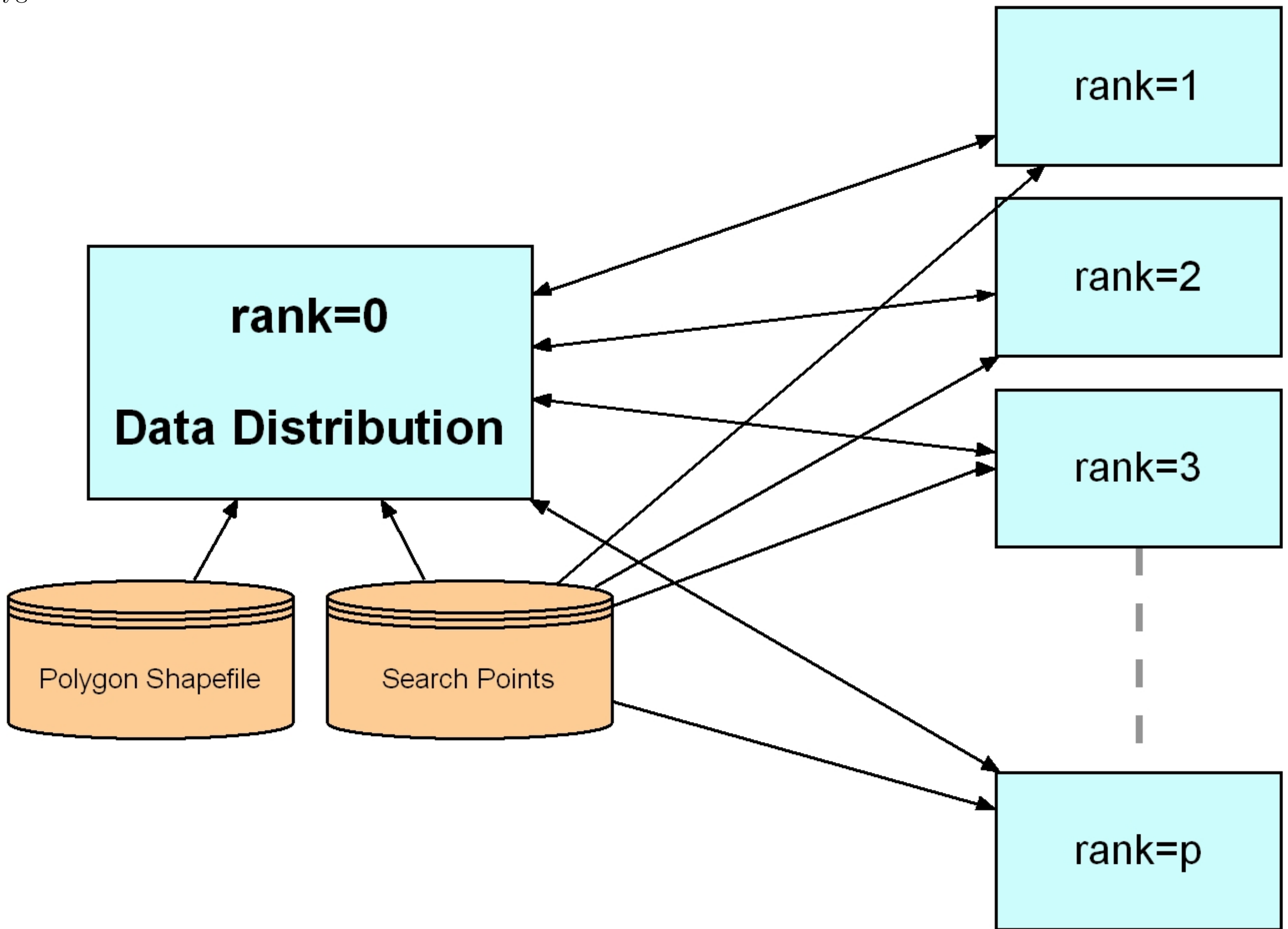


0.2 Decomposing problem to multiple CPUs

The task of determining what polygon multiple points fall within is classified as an embarrassingly parallel problem. It is classified this way because there are many unrelated points to test and there is very little inter-CPU communication required.

While designing the parallel version of the program I contemplated how I wanted to distribute data. I determined that I wanted to use MPI for distribution of search points and the availed Network File System directories for distribution of the polygon data to the other nodes.

All of the nodes would be using the same polygon data and it would only have to be loaded once per run. Because the polygon data is the same at all nodes it would take the same time to load everywhere. On the hpc-class machine used there is a temporary directory (/ptmp) that is mounted on all nodes and would therefore be suitable for storing the polygon data.



0.2.1 Using MPI to move search point data

As discussed above, the search points are read in on the rank=0 node and then distributed to non-root nodes using MPI.

Each search point is stored within the program as a C structure:

```
typedef struct PT {
   double x;
   double y;
   char source[11];
} POINT;
```

It would be possible to use MPI derived data types for encapsulating the structure and sending it with MPI. This technique suffers from poor performance and increased complexity for the programmer. Instead I recognized that the three members of the POINT structure are all intrinsic data types and not pointers to other memory. This fact, along with C's memory allocation rules, make sure that the POINT structure and its elements will be located in contiguous memory. A sequence of binary bytes can be handled with MPI by referring to it as type MPI_BYTE.

Sending the structure is then simply a matter of determine it's size. Typically the size of of a C variable is determined by using the sizeof() macro. One pitfall of using sizeof() is that it takes into account word alignment. Word alignment is require because most modern computers can only access memory a word at a time instead of a byte at a time. On a 32-bit machine calling sizeof(POINT) gives a value of 32, even though the actual number of bytes required is 27.

Using MPI_BYTE do send binary is not usually an acceptable practice because MPI is designed to work for heterogeneous clusters where not all machines are the same. There would be incompatibility of binary data sent between 32 bit machines and 64 bit machines or on little endian (Intel) versus big endian (Motorola, Digital, Sun, ect) machines. Since most purpose-built clusters these days, including hpc-class, are homogeneous in nature this is not an issue.

Once I had determined how to send and receive the search point data using MPI I concentrated on finding the highest performance technique for spreading the data cross the available nodes.

The first technique that I implemented and test was to using block sends and a for loop to send search points to each node, one by one. This technique was wasteful because the running time of the polygon searching algorithm is not constant. The result is that some nodes would finish faster than others and would spend significant time waiting for the more data.

After eliminating blocking sends I experimented with using MPI buffered immediate sends for data distribution. To use buffered sends I had to allocate buffer space on the master node using MPI_Buffer_attach. The buffer allocated had to be large enough to hold all of the data I intended on sending or it needed to keep track of how much of the buffer was used. I implemented both techniques and found that they were both a pain to implement and wasteful of memory. The performance with using immediate buffered sends was the same as the blocking sends anyhow.

I was puzzled when I determined that the behavior of the program using blocking sends was quite similar to the program using buffered immediate sends and waits. I spent a significant chunk of time trying to account for this behavior. I ended up writing a small test program where I could introduce a delay on the receiving side so I could determine the behavior of the MPI_Send and the MPI_Isend.

```
if ( 0 == rank ) {
   /* root node */
   MPI_Issend(&i,1,MPI_INT,1,0,MPI_COMM_WORLD,&request);
   printf("# done with immediate send\n");
   fflush(stdout);
   MPI_Wait(&request,MPI_STATUS_IGNORE);
   printf("# done waiting\n");
   fflush(stdout);
} else {
   sleep(4);
   MPI_Recv(&i,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
}
```

Using the test program above I determined that the MPICH / Myricom MPI implementation used buffered sends as it's basis for the standard MPI_Send. This was observed by nothing that even with a four second delay before the receiving process posted a MPI_Recv the MPI_Wait would still immediately say that the send was done.

I finally discovered that using MPI_Issend would allow me to do an immediate send and be able to poll, using MPI_Wait, to see when it was received. By being able to check if a node was done I was able to implement an effective means of load balancing between the nodes. I wrote a function for sending data that would check to see if the previous data point had been received by a node and if it had then send it a new data point. If the node was node yet done the program would go on to the next node and perform the same check. Below is the relevant section of my program that implements this.

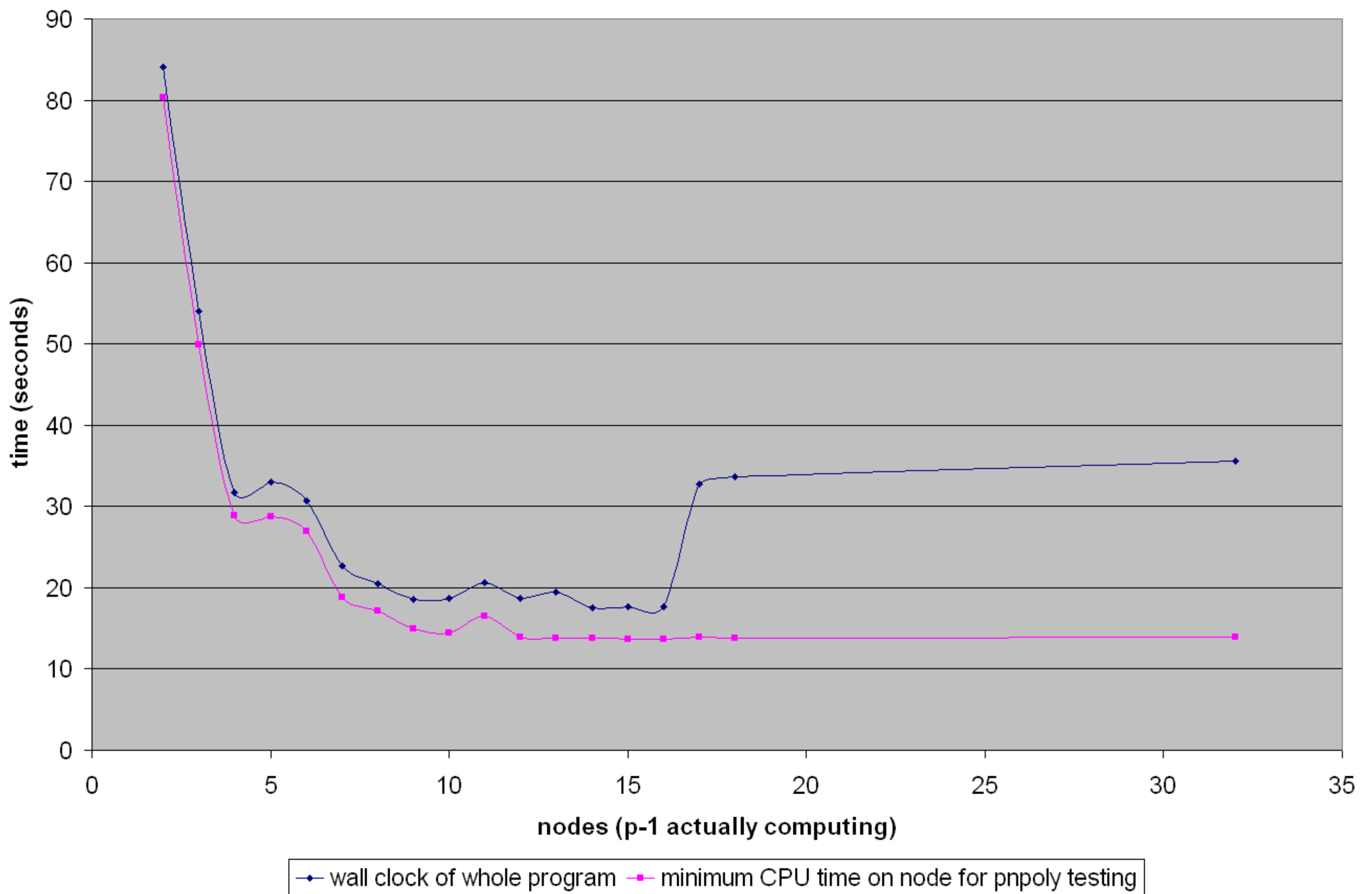
```
for ( ; ; ) {
   j=(i%(p-1))+1;
   i++;

   /* see if our last isend is done */
   MPI_Test(&request[j],&flag,MPI_STATUS_IGNORE);

   if ( flag ) {
```

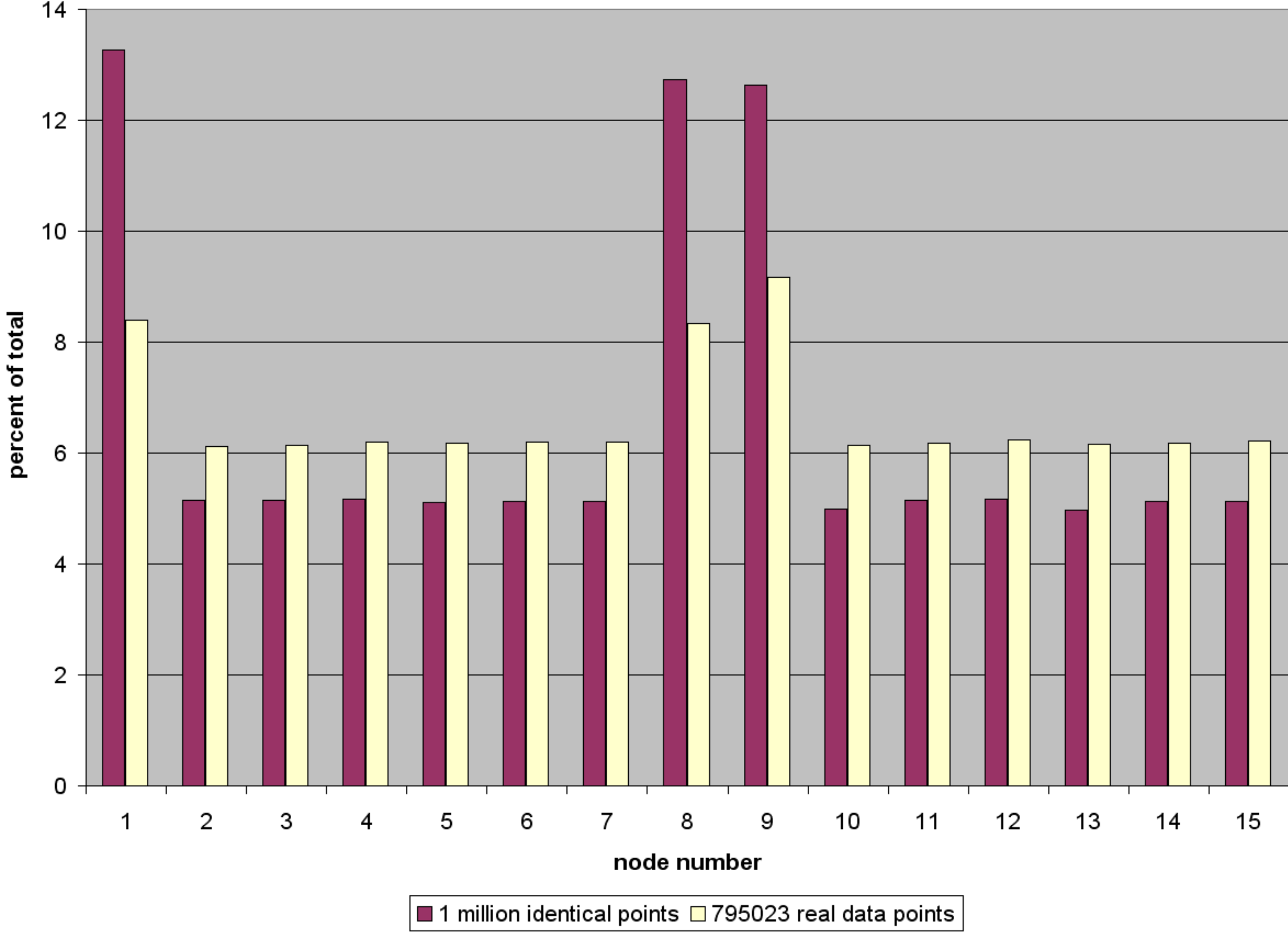
```
        /* send our point one piece at a time */
        return MPI_Issend(pt,sizeof(POINT),MPI_BYTE,j,tag,MPI_COMM_WORLD,&request[j]);
    }
}
```

The result of this technique is that each CPU is kept at nearly fully load. A slow search on one node has no effect on another node because the program will simply pass by busy nodes and send the next search point to the next available node.

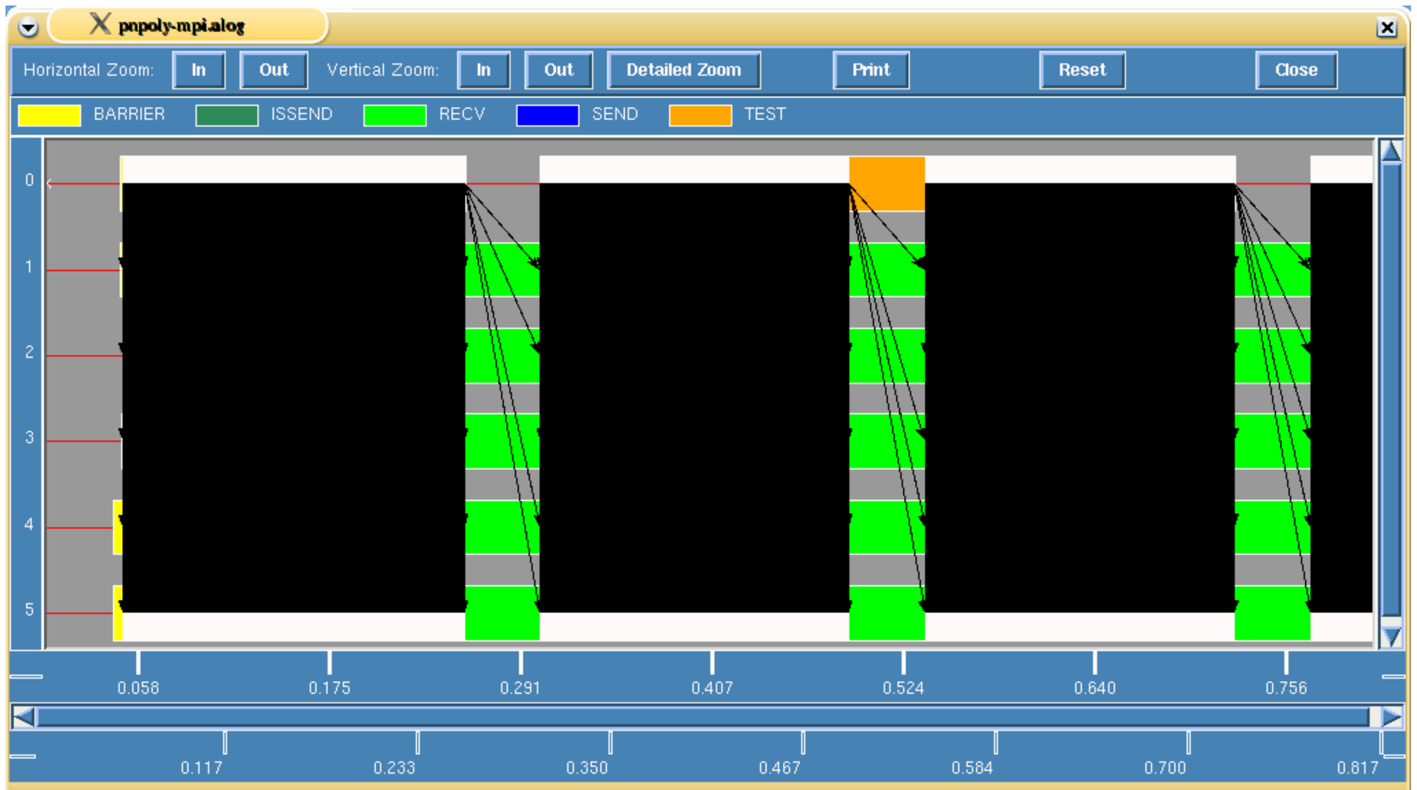
0.3 Results and Conclusions

I ran my program on hpc-class using the bmpi run command. I took to primary timing measurements, wall clock time, and CPU time on each node, to characterize performance with different numbers of nodes. I did not make any attempt to flush the cache or otherwise manipulate the system because I wanted to gather real world performance data.





Percent of points handled by node
percent of total
0
2
4
6
8
10
12
14
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
node number
1 million identical points
795023 real data points



I was not able to run my program on the Cray T3E machine at the Pittsburgh Supercomputer Center because of incompatibilities between the Cray and the Shapelib library and because I did not have enough disk space available on the Cray. The Shapelib library assumes that it will have 32-bit integers and uses this assumption throughout its code to manipulate binary data. I attempted to fix this problem by using C's typedef functionality to map an int32 to the system type short, but this did not solve the problem. After conferring with the author of Shapelib, Frank Warmerdam, I concluded that making Shapelib run on the Cray would be beyond the scope of this project.

My attempt at parallelizing shpdump-poly was successful. As shown in the graphs above I found that the program ran faster with more processors until Amdahl's law catches up.

0.4 Future Plans

I would like to optimize further the serial portion of my code to increase performance. In particular there is a function that translates the shape ID's in human readable labels that takes a long time to run. It may be possible to do this on the root node while sending test points for an overall savings in time. Also it may be possible to use pthreads or Open MP to more effectively take advantage of the dual processor nodes.

Besides the polygon searching program I also have a program that finds the nearest road to a point. I would like to trying parallelizing this program in hopes of improving its performance. This program makes heavy use of I/O as it loads and unloads data totalling about 20 gigabytes. I believe that a cluster would need very good disk I/O performance to be able to handle this job efficiently - something that hpc-class does not have available since it uses NFS.

.1 Appendix A - Source Code

```
#include "shapefil.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <math.h>

/* MPI stuff */
#include "mpi.h"

#include <unistd.h>
#include <sys/stat.h>

/* our search point */
typedef struct PT {
        double x;
        double y;
        char source[11];
} POINT;

/* our individual shapes */
typedef struct {
        SHPObject       *o;
        int     value;
        int     hitCount;
        int     record;
}       SHPstr;

int dbfDisplayColumns[16];
int dbfDisplayColumnsCount=0;
double atof();


/* Returns true if structure becomes filled with a valid lat / lon pair */
int getTestPoint(POINT *search, FILE *fp) {
        char buffer[256];

        if ( (0 != fgets(buffer,256,fp)) ) {
                if ( 3 == sscanf(buffer,"%lf %lf %10s",&search->y,&search->x,search->source) ) {
                        return 1;
                }
        }
        search->x=search->y=0;
        strcpy(search->source,"");
        return 0;
}

int mpi_send_point(POINT *pt, int dest, int tag ) {
```

```
        return MPI_Send(pt,sizeof(POINT),MPI_BYTE,dest,tag,MPI_COMM_WORLD);
}

int mpi_send_point_next_ready(POINT *pt, int tag, int p, int cleanup) {
        int retval;
        static int i=0;
        int j=1;
        static int k=1;
        int flag;
        static int count=0; /* keep track of if we are initialized */
        static int points=0;

        MPI_Status status;
        static MPI_Request *request;
        static int *sentCount;

        if ( cleanup ) {
                double percent;
                fprintf(stdout,"# Data sent to each node (%d total):\n",points);
                for ( j=1 ; j<p ; j++ ) {
                        fprintf(stdout,"# [%d] %07d points (%02.2lf%% of total)\n",j,sentCount[j],(double) (
                }
                /* free our array of request handles */
                free(request);
                free(sentCount);
                return 0;
        }

        if ( 0 == count ) {
                /* first time through */
                /* allocate memory for p number of MPI_Request variables */
                request = (MPI_Request *) malloc(p*sizeof(MPI_Request));
                sentCount = (int *) calloc(p,sizeof(int));
                count=1;
        }


        if ( k<p ) {
                retval = MPI_Issend(pt,sizeof(POINT),MPI_BYTE,k,tag,MPI_COMM_WORLD,&request[k]);
                sentCount[k]++;
                k++;
                return retval;
        }


        for ( ; ; ) {
                j=(i%(p-1))+1;
                i++;

                /* see if our last isend is done */
                flag=0;
                MPI_Test(&request[j],&flag,MPI_STATUS_IGNORE);
```

```
                if ( flag ) {
                        sentCount[j]++;
                        points++;
                        /* send our point one piece at a time */
                        return MPI_Issend(pt,sizeof(POINT),MPI_BYTE,j,tag,MPI_COMM_WORLD,&request[j]);
                }
        }
        return -1;
}


int mpi_get_point(POINT *pt, int source, int tag) {
        return MPI_Recv(pt,sizeof(POINT),MPI_BYTE,source,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
}

/*
Point in polygon test

Input parameters:
npol               : number of vertices
xp[npol], yp[npol] : x,y coord of vertices
x,y                : coord of point to test

Return Value:
0 : test point is outside polygon
1 : test point is inside polygon

Notes:
 if test point is on the border, 0 or 1 is returned.
 If there exists an adjacent polygon, the point is
 _in_ only in one of the two.
*/
int pnpoly(int npol, int start, double *xp, double *yp, double x, double y) {
        int i, j, c = 0;

        for (i = start, j = npol-1; i < npol; j = i++) {
                if (    ( ((yp[i]<=y) && (y<yp[j])) || ((yp[j]<=y) && (y<yp[i]))) &&
                        (x < (xp[j] - xp[i]) * (y - yp[i]) / (yp[j] - yp[i]) + xp[i])) {
                        c = !c;
                }
        }
        return c;
}

/* shapes can consist of multiple polygons ... this routine breaks multi-part shapes into individual polygons

int pnpolyMulti(int npol, double *xp, double *yp, int nParts, int *panPartStart, double x, double y) {
        int i;

        for ( i=0 ; i<nParts-1 ; i++ ) {
                int vertices = panPartStart[i+1] - panPartStart[i];

                if ( pnpoly(vertices,panPartStart[i],xp,yp,x,y) ) {
```

```
                        return 1;
                }
        }
        /* processes the last part */
        if ( pnpoly(npol-panPartStart[i],panPartStart[i],xp,yp,x,y) ) {
                return 1;
        }

        return 0;
}

/* compare two shapes by presort value*/
int SHPstrcmp_value( SHPstr *a, SHPstr *b) {
        return  b->value - a->value;
}
/* compare two shapes by record number */
int SHPstrcmp_record( SHPstr *a, SHPstr *b) {
        return  b->record - a->record;
}

/*
our initial test is to see if our point falls within the bounding box
*/
inline int bbox(double x, double y, SHPstr *s) {
        if ( (x >= s->o->dfXMin && x <= s->o->dfXMax && y <= s->o->dfYMax && y >= s->o->dfYMin ) ) {
                return 1;
        }
        return 0;
}


void printShapeHits(DBFHandle hDBF, SHPstr *s) {
        int j;
        printf("%d|",s->hitCount);
        for ( j=0 ; j<dbfDisplayColumnsCount ; j++ ) {
                printf("%s|",DBFReadStringAttribute( hDBF,s->record, dbfDisplayColumns[j] ));
        }
        (void) putchar('\n');
}

int main( int argc, char **argv ) {
        /* shapefile */
        char shapefileName[256];
        SHPHandle hSHP;
        SHPstr *s;
        SHPstr x;
        int nShapeType, nEntities, i;
        double adfMinBound[4], adfMaxBound[4];

        /* dbase */
        DBFHandle hDBF;
        int presortColumn=0;
```

```
/* point in polygon */
int totalVerticies=0;
POINT search;
int found;
int j;

/* options */
int split=0;
int verbose=0;
int presort=0;
int immediateDisplay=0;
int resort=0;
int skipBboxTest=0;

/* timing */
clock_t c_begin, c_end;

/* MPI stuff */
int rank;
int p;
FILE *fp;
int points;
char inFileName[256];
struct stat file_stat;
int *aHit;

MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);


/* initialize structure */
search.x = search.y = 0.0;
strcpy(search.source,"");

while ((j = getopt (argc, argv, "bhp:ris:vf:")) != -1) {
        switch (j) {
                case 'h':
                        if ( 0 == rank ) {
                                fprintf(stdout,"-b          Disable bounding box test.\n");
                                fprintf(stdout,"-f          Read points from file not STDIN.\n");
                                fprintf(stdout,"-h          This help.\n");
                                fprintf(stdout,"-i          Immediately display result.\n");
                                fprintf(stdout,"-p          Sort polygons before running.\n");
                                fprintf(stdout,"-r          Re-sort polygons after each search.\n");
                                fprintf(stdout,"-sDIRECTORY Split input to one file per polygon.\n");
                                fprintf(stdout,"-v          Verbose mode.\n");
                        }
                        exit(EXIT_SUCCESS);
                case 's':
                        fprintf(stdout,"# Split Mode: creating/appending one file per polygon\n");
                        if ( 0 == strcmp("",optarg) ) {
```

```
                                fprintf(stdout,"Split mode: need input directory!\n");
                                exit(EXIT_FAILURE);
                        }

                        split=1;
                        if ( chdir(optarg) ) {
                                fprintf(stdout,"Could not change to %s. Quitting.\n",optarg);
                                exit(EXIT_FAILURE);
                        }
                        break;
                case 'f':
                        strncpy(inFileName,optarg,128);
                        if ( 0 != stat(inFileName,&file_stat) ) {
                                fprintf(stdout,"# Cannot open test point file: %s\n",inFileName);
                                goto out;
                        }
                        if ( 0 == rank )
                                fprintf(stdout,"# Root node will read search points from: %s\n",inFil
                        break;
                case 'b':
                        if ( 0 == rank )
                                fprintf(stdout,"# Bounding box test disabled.\n");
                        skipBboxTest=1;
                        break;
                case 'v':
                        if ( 0 == rank )
                                fprintf(stdout,"# Verbose Mode\n");
                        verbose=1;
                        break;
                case 'p':
                        presort=1;
                        presortColumn = atoi(optarg);
                        if ( 0 == rank )
                                fprintf(stdout,"# Pre-sorting polygons on column %d\n",presortColumn)
                        break;
                case 'r':
                        if ( 0 == rank )
                                fprintf(stdout,"# Countinuosly resorting polygons\n");
                        presort=1;
                        break;
                case 'i':
                        if ( 0 == rank )
                                fprintf(stdout,"# Immediate display mode\n");
                        immediateDisplay=1;
                        break;
        }
}

if ( p > 0 && 0 == strcmp(inFileName,"") ) {
        fprintf(stdout,"# -f option needed when more than one node requested\n");
        goto out;
}
```

```
if ( p == 1 ) {
        fprintf(stdout,"# MPI version needs more than one node!\n");
        goto out;
}

if ( argc < optind+2 ) {
        fprintf(stdout,"%s: shapefile [-p presort_column] [-s split_directory] dbfDisplayColumn [dbfD
        exit(EXIT_FAILURE);
}

if ( 0 == rank ) {
        printf("# Program run as: ");
        for ( i=0 ; i<argc ; i++ ) {
                printf("%s ",argv[i]);
        }
        printf("\n");
}

/* get our shapefile name */
strncpy(shapefileName,argv[optind++],256);

/* get all of the display columns */
dbfDisplayColumnsCount=0;
for ( ; optind<argc ; optind++ ) {
        dbfDisplayColumns[dbfDisplayColumnsCount] = atoi(argv[optind]);
        dbfDisplayColumnsCount++;
}

/* Open the passed shapefile. */
hSHP = SHPOpen( shapefileName, "rb" );
if( hSHP == NULL ) {
        printf( "SHPOpen(\"%s\")\n", shapefileName );
        exit(EXIT_FAILURE);
}

/* Open DBF file associated with shapefile. */
hDBF = DBFOpen( shapefileName, "rb" );
if( hDBF == NULL ) {
        printf( "DBFOpen(%s,\"r\") failed.\n", shapefileName );
        exit(EXIT_FAILURE);
}
/* Warn user that DBF file doesn't contain anything */
if( DBFGetFieldCount(hDBF) == 0 ) {
        printf( "There are no fields in this table!\n" );
        exit(EXIT_FAILURE);
}

/* Print out the file bounds. */
SHPGetInfo( hSHP, &nEntities, &nShapeType, adfMinBound, adfMaxBound );


/* Check to makesure we are dealing with a polygon shapefile */
```

```
if ( nShapeType != SHPT_POLYGON && nShapeType != SHPT_POLYGONZ && nShapeType != SHPT_POLYGONM) {
        printf("Not a polygon File!\n");
        exit(EXIT_FAILURE);
}


/* only print startup information if we are root node */
if ( 0 == rank ) {
        printf("# Shapefile %s\n",shapefileName);
        printf("# Displaying columns {");
        for ( i=0 ; i<dbfDisplayColumnsCount-1 ; i++ ) {
                printf("%d,",dbfDisplayColumns[i]);
        }
        printf("%d}\n",dbfDisplayColumns[i]);

        fprintf(stdout, "# File Bounds: (%12.3f,%12.3f,%lg,%lg)\n"
                        "#          to  (%12.3f,%12.3f,%lg,%lg)\n",
                                adfMinBound[0],
            adfMinBound[1],
            adfMinBound[2],
            adfMinBound[3],
            adfMaxBound[0],
            adfMaxBound[1],
       adfMaxBound[2],
            adfMaxBound[3] );
}

/* allocate a two-dimensional array of integer nEntities large so we can
send our hit counts back to the root node
array will be nEntities rows by 2 columns
*/
aHit = (int *) malloc(nEntities * sizeof(int));

/* time our initial load */
c_begin = clock();

/*      Load an array with pointers to all the shapes.  */
s = calloc( (size_t) nEntities,sizeof(SHPstr));
if ( NULL == s ) {
        fprintf(stdout,"calloc failed\n");
        exit(EXIT_FAILURE);
}

for ( i=0; i < nEntities; i++ ) {
        s[i].record = i;
        s[i].hitCount=0;
        s[i].o = SHPReadObject( hSHP, i );
        s[i].value = atoi(DBFReadStringAttribute( hDBF, i, presortColumn ));

        totalVerticies += s[i].o->nVertices;
}

if ( rank > 0 ) {
```

```
		/* quick sort the polygons based on sort criteria before running */
		if ( presort ) {
			if ( 0 == rank )
				fprintf(stdout,"# Creating initial search order ... ");
			qsort(s,nEntities,sizeof(SHPstr),SHPstrcmp_value);	/* sort the polygons first */

			if ( 0 == rank )
				fprintf(stdout," done sort\n");
		}
	}

	/* stop our inital ending */
	c_end = clock();
	fprintf(stdout,"# Node %d: %d verticies and %d polygons loaded into memory in %2.4lf seconds.\n",rank

	c_begin = clock();
	if ( 0 == rank ) {
		/* root node */



		/* we open out input file on the root node and send it from there */
		fp = fopen(inFileName,"r");


		/* syncronize */
		MPI_Barrier(MPI_COMM_WORLD);

		/* read points and distribute evenly to nodes */
		for ( points=0 ; getTestPoint(&search,fp) ; points++ ) {
//			j=(points%(p-1))+1;
//			mpi_send_point(&search, j, 0);
			mpi_send_point_next_ready(&search, 0,p,0);


		}
		/* tell it to clean up and free it's dynamically allocated memory */
		mpi_send_point_next_ready(&search, 0,p,1);

		fclose(fp);
		printf("# root node sent %d points to %d nodes\n",points,p-1);


/* set our point to (0,0) and that signals the non-root processes to finish up */
		search.x=search.y=0;
		for ( i=1 ; i<p ; i++ ) {
			mpi_send_point(&search, i, 0);
		}
		printf("# done sending\n");


		/* see comment below for why we need this sort */
		qsort(s,nEntities,sizeof(SHPstr),SHPstrcmp_record);
```

```
	for ( j=1 ; j<p ; j++ ) {
		MPI_Recv(aHit,nEntities,MPI_INT,MPI_ANY_SOURCE,2,MPI_COMM_WORLD,&status);

		/* add on the other node's counts */
		for ( i=0 ; i<nEntities ; i++ ) {
			s[i].hitCount += aHit[i];
		}
	}

	/* Now we have all of results back to root node, print and look them up */
	printf("# Results\n");
	for ( i=0 ; i < nEntities ; i++ ) {
		if ( s[i].hitCount ) {
			printShapeHits(hDBF, &s[i]);
		}
	}



} else {
	/* non-root nodes */
	/* syncronize */
	MPI_Barrier(MPI_COMM_WORLD);

	for ( points=0 ; ; points++ ) {
		mpi_get_point(&search, 0, 0);

		/* if latitude and longitude are zero than we drop out */
		if ( 0==search.x && 0==search.y ) {
			break;
		}

		/* we have point */
		for ( i=0,found=0 ; ( i<nEntities ) ; i++ ) {
			if ( ( skipBboxTest || bbox(search.x,search.y,&s[i]) ) && pnpolyMulti(s[i].o-
				/* Mark the hit */
				s[i].hitCount++;
				found=1;

				/* immediately print to screen */
				if ( immediateDisplay ) {
					printShapeHits(hDBF, &s[i]);
				}

				/* Append to a file named the dbfDisplayColumn result */
				if ( split ) {
					FILE *split_fp;
					char filename[128];

					strncpy(filename,(void *) DBFReadStringAttribute(hDBF,s[i].re

					split_fp = fopen(filename,"a");
```

```
                                        if ( NULL == split_fp ) {
                                                fprintf(stdout,"Error writing split file: %s\n",filen
                                                exit(EXIT_FAILURE);
                                        }
                                        fprintf(split_fp,"%lf\t%lf\t%s\n",search.y,search.x,search.so
                                        (void) fclose(split_fp);
                                }

                                /* Each time we get a hit we move the shape up in
                                 priority so it gets searched sooner */
                                if ( resort && i > 0 ) {
                                        x=s[i-1];
                                        s[i-1]=s[i];
                                        s[i]=x;
                                }

                        }
                }
        }
        /* stop timing */
        c_end = clock();
//              fprintf(stdout,"# Node %d spent %lf seconds testing %d points.\n",rank,(float) (c_end-c_begin
        fprintf(stdout,"# [%d] spent %lf seconds.\n",rank,(float) (c_end-c_begin) / CLOCKS_PER_SEC );

        /* we are now ready to send our hit counts back to the root node. If we
        used continuous re-sorting our order of s will be different, so we
        now resort based on our s[].record which will give us the right order
        on all machines
        */
        qsort(s,nEntities,sizeof(SHPstr),SHPstrcmp_record);

        for ( i=0 ; i<nEntities ; i++ ) {
                aHit[i]=s[i].hitCount;
        }
        /* send our results back to root node */
        MPI_Send(aHit,nEntities,MPI_INT,0,2,MPI_COMM_WORLD);


    }


    /* free dynamically allocated memory */
    free(s);
    free(aHit);
    SHPClose( hSHP );
    DBFClose( hDBF );

out:
    /* finish our MPI communications */
    MPI_Finalize();

    return 0;
}
```