CS 651 Parallel Computation - Fall 1992

Sequential Image Convolution and Spot Finding

Homework #1

Matt Judd, John Karpovich, Emily West

Problem Description:

The goal of this assignment was to implement a sequential image convolver and spot finder. In the future, this code will be ported and parallelized for a number of different parallel machine architectures.

Approach:

Portions of code from the dppx.c source were used in reading in the image file. Additionally, the DD_chararray class defined in app_misc.o was used to implement the image matrix in the image class. An attempt was made to design classes that would eventually lend themselves well to parallelization; the success of this design feature will be assessed later.

The following classes were developed:

- **Image** - This class implements the data structure and operations necessary to input an image file in ppx format, convolve the image with supplied filters, create the image matrix produced by the application of the double derivative x and y filters, perform the spot finding algorithm, and output the resultant image in ppx format. Input and output of images can be accomplished by either regular files, or stdin and stdout.
- **FilterList** - This class implements a data structure which contains a list of filter structures.This data structure includes the name of the file which contains the filters to be placed in the filter list, as well as functions to get the filters, return a single filter, and return the size of the largest row and column in the filter list.
- **Point** - This class implements the data structure and operations necessary to manipulate a single cartesian (x,y) point. This class includes functions and operators for assignment, printing, equivalence checking, etc. The implementation of this class was a trade-off. While gains in readability and code simplicity were attained, speed of execution was reduced from the overhead involved in the class implementation.
- **Spot** - This class implements a single spot found in the image. Included in the definition are a list of pixels in the spot (a pointList - see below) and the extreme x & y values of all of the pixels (possibly to be used later in problem decomposition and scheduling).
- **PointList & SpotList** - These classes implement the lists needed to store and manipulate lists of points and spots respectively. The pointList class contains the number of points in the list (possibly needed later for problem decomposition), the coordinates of the point, and a pointer to the next point in the linked list. The spotList class is very similar and contains the number of spots in the list, the spot itself and a pointer to the next spot in the linked list. List operations include adding to the front of the list, destructively removing and returning the first element of the list, copying the list, checking for empty, checking if an element is contained in the list, and other minor features. These operations allow

the lists to be used as stacks when necessary or as regular linked lists in other cases.

Problems Encountered:

The DD_chararray class is deficient in the manner that access is given to an individual element of the matrix. This operation requires a get_r_ptr() call and an add (see the overload of [] in the image class). Thus, using the operator[] is very expensive in the convolve routine. Consequently, our code was modified to avoid use of this operator. A greater efficiency in the DD_chararray class would be gained if access to the matrix were less expensive. In our application of convolve, we use pointers to the image matrix to avoid the need of calling the operator[] (and thus the get_r_ptr() function) for every point access.

Observations:

While observing individual runs of the program, we observed that convolving the image with the supplied filters, and performing the double derivative step (essentially two more convolves), took much longer than performing the spot finding algorithm. While this is in no way a scientific observation, our intuition is that parallelization of the algorithm has the potential for good speedup. This is a result of the fact that the data parallel nature of the convolution portion is more conducive toward parallelization them the current spot finding algorithm. The spot finding algorithm will possibly require the extraction of functional parallelism or, a "bag of tasks" approach where a task will most likely be composed of multiple "possible spots".

Another observation is that in our spot finding algorithm, we are currently determining whether a point is in the check point list with a linear scan. This is slow, especially with large spots. Another way to do this would be by creating another image of equal size, with each point set to zero. As each point in the original list is added to the point list, change its value in the new image to 1. Checking whether a point is in the list can be done by checking that points value in the new list. While this uses more memory, it seems likely that it would result in some amount of speedup.

When we go to parallelize there will be many ways that we can change the implementation to improve the granularity of the application to be either coarse or fine so that performance will be good on the target architecture. Additionally, we are considering is using Tom Olson's blob coloring algorithm to find all four-connected spots, and then parcel these out to separate processors to perform the cavity search and the spot coloring algorithms. A profile of the code using gprof (included) indicated that most of the time was spent doing multiplies and divides. This obviously points to parallelization of the convolution step.

As a caveat to the future parallelization, it may be beneficial to acquire other images in which the distribution of found spots is not so regular. The test image provided resulted in a very uniform distribution of spots in the final image. This would indicate that simply carving the convolved image into regular, equal pieces would do almost as well, and maybe better due to the overhead of a heuristic that partitioned the image based on spot density for this particular image.

An interesting observation is that in the final image the spots are colored with consecutive numbers, and since there are far more spots than 255, there is a "wave" effect in the final coloration of the spots. This simply confirms that our numbering scheme is correct. The spots numbered below a certain threshold, probably around 80 are probably not distinguishable because of grey-

scale.

Performance:

All performance tests were run on sparcStation2 machines and with code compiled using the -O optimization flag. The image file used was the 512x512 a00661.ppx file provided in the ppx library. Our goal for performance testing was to see the relative effect of varying problem size (in our case the amount of pre-filtering done) and the effect of choosing one algorithm over another.

After we first wrote our algorithm, we realized that the development of the core image could be done all in one pass, instead of convolving the image separately two times followed by a synthesis step which we had originally implemented. Since we had already put the code in place, we decided to run a comparison (we used #define statements to keep the code separate). Head-to-head, the cpu times (as measured by the time command) for the "fast" algorithm vs the "slow" one were 2.9 seconds vs 3.5 seconds using a 5x5 Gaussian smoother filter only. Using a multi-stage filter (the 5x5 Gaussian followed by a crude 3x3 edge detector filter) resulted in times of 3m10 seconds and 4 minutes for the slow and fast algorithms respectively. This result is consistent with expectations - a roughly constant factor (for the given image size) speed-up.

To test the effects of problem scale, the results from above between the Gaussian only and the multi-filter runs can be used. We would expect a slow down from one run to another proportional to the total number of pixels in the convolution masks. For the Gaussian filter, there are 5x5 = 25 total convolution points while there are 5x5 + 3x3 = 34 total points when both the Gaussian and edge detection filters are used. There is a fixed initialization cost to load the image and filters. Additionally, the double derivative and spot finding costs should not vary significantly since neither filter greatly reduces the number of spots found. So the times of the two runs should be something like Fixed + totalConvulutionPixelsxC = time, where C is some fixed time for doing each mask pixel worth of convolution. Therefore, the times should be linear, which they appear to be (though time constraints have not allowed us to collect and plot the data properly).

Pointers to Code and Result Images:

The code can be found in ~jfk3w/cs651/hw1/code.

convolve.c

pointList.h

The result images can be found in ~jfk3w/cs651/hw1/images.

gaussedge.ppx

gauss.ppx

The executable is ~jfk3w/cs651/hw1/convolve.

```
/*
 * Parallel Genetic Algorithm for dynamic scheduling.
 *
 * CS651 Parallel Computation Final Project
 * Fall 1992
 *
 * Emily West
 *
 * pgaslave.c - worker for pga
 *
 * PGA code based on code written by Bruce Childers, Charlie Viles, and
 * Emily West Spring 1992 - GA for static scheduling.
 *
 * The following code implements a Parallel Genetic Algorithm
 * for dynamic Scheduling. We refer the reader to the seminal
 * project writeup for all of the details.
 * PVM is used for the parallelization.
 *
 *      COMPILE:        make pgaslave
 *      OBJECTS:        pgaslave.o
 *                      error.o util.o pgraph.o sched.o chrom.o
 *                              proc.o population.o
 *      HEADERS:        defs.h
 *                      chrom.h error.h pgraph.h population.h proc.h
 *                              sched.h util.h
 *                      pvmuser.h
 *
 *
 *      MODIFICATIONS:
 *              Dec 7 - Added -DSEQUENTIAL FLAGS. pgaslave can now be compiled
 *                      as either a standalone sequential GA called with
 *                      command line parameters, or as a slave from pgamaster
 *                      in which case it receives necessary data from the
 *                      master.
 *              Dec 10 - added child creation - modified population
 *                      mechanics maintenance.
 *              Dec 11 - added epoch exchanges,
 *                       added population selection mechanisms
 *                              1 - probabalistically / no replacement
 *                              2 - equilikely / no replacement
 *              Dec 12 - performance tuning
 */

#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <error.h>
#include <pgraph.h>
#include <proc.h>
#include <sched.h>
#include <util.h>
#include <population.h>
#ifndef SEQUENTIAL
#include <pvmuser.h>
#endif
#include <defs.h>

extern struct population *pickSubPop(struct population *P);

/* Global Declarations */
int nt;
int np;
int popSize;
int newpopSize;
int subpopSize;

#ifdef SEQUENTIAL
int parseArgs (int argc, char **argv, char *procFile, int *numGens,
               int *numExch, double *XRate, double *EOMutRate,
               double *PAMutRate, char *bestFile)
{
    double atof();

    if (argc < 10) {
        printf ("Usage: %s <file for processors> <popSize> <subpopSize>\n",
                argv[0]);
        printf ("          <numGens> <numExch> <XRate> <EOmutRate>\n");
        printf ("          <PAmutrate> <best scores file>\n");
        exit (1);
    }

    strcpy (procFile, argv[1]);
    popSize = atoi (argv[2]);
    subpopSize = atoi (argv[3]);
    *numGens = atoi (argv[4]);
    *numExch = atoi (argv[5]);
    *XRate = atof (argv[6]);
    *EOMutRate = atof (argv[7]);
    *PAMutRate = atof (argv[8]);
    strcpy (bestFile, argv[9]);
}
#endif

void main(int argc, char *argv[])
{
    /* Genetic Algorithm Variables */
    struct graph *G;            /* precedence graph */
    struct proc_set *PS;        /* the processor set */
    struct schedule *S, *best;  /* current best schedule */
    struct population *P;       /* current population of schedules */
    struct population *EP;      /* epoch population of migrant schedules */
    struct population *SP;      /* sub population of migrant schedules */
    struct edgelist *edges;
    int numGens;                /* number of generations to run */
    int numEpochs;              /* number of subpopulation exchanges */
    int ne;
    long clock;                 /* random number generator seed */
    long pid;
    double XRate;               /* Crossover rate */
    double EOMutRate;           /* mut rate in the execution ordering */
    double PAMutRate;           /* mut rate in the processor assign. */
    struct timeval tp;          /* for getting the start time */
    char bestFile[80];          /* file where best schedules are saved */
    char procFile[80];
    FILE *fp, *fopen();         /* file variables */

    /* PVM management variables */
    char component[9];          /* name of this component */
    int inst;                   /* instance of this component */
    int master_inst;            /* instance of the master */
    int ngrp;                   /* number of neighbors */
    int *neighbor_list;         /* instance numbers of neighbor slaves */

    /* General Variables */
    int i, j, e;                /* index variables */

/****************** Initialization ******************/
```

```
    /* seed random number generator */
    pid = (long)getpid();
    srand48(pid);

    /* get start time */
    gettimeofday(&tp, NULL);

#ifndef SEQUENTIAL
    /* Enroll the application */
    strncpy(component, "pgaslave", 9);
    inst = enroll(component);

/*    printf("SLAVE: Comp %s Inst %d\n", component, inst); fflush(stdout);*/

    /* recieve neighor info */
    rcv(NEIGHBOR_INFO1);
    getnint(&master_inst, 1);
    getnint(&ngrp, 1);

    neighbor_list = (int *)malloc(ngrp * sizeof(int));
    rcv(NEIGHBOR_INFO2);
    getnint(neighbor_list, ngrp);

    /* receive computation info */
    rcv(COMPUTATION_INFO);
    getnint(&nt, 1);
    getnint(&np, 1);
    getnint(&popSize, 1);
    getnint(&subpopSize, 1);
    getnint(&numGens, 1);
    getnint(&numEpochs, 1);
    getndfloat(&XRate, 1);
    getndfloat(&EOMutRate, 1);
    getndfloat(&PAMutRate, 1);
    G = unpack_graph();
    PS = unpack_procset();
#else
    /* Get the processor graph and generation info */
    parseArgs (argc, argv, procFile, &numGens, &numEpochs,
               &XRate, &EOMutRate, &PAMutRate, bestFile);

    /* create and read the processor info */
    if ((fp = fopen(procFile, "r")) == NULL)
        error (MALLOC_ERR, "main", "opening processor file", FATAL);
    PS = readProcessors (fp);
    fclose (fp);

    /* create and read the precedence graph */
    G = readGraph(stdin);
    edges = G->E;
    ne = G->ne;
#endif

/****************** Computation ******************/

    /* create, initialize and evaluate the initial population */
    newpopSize = popSize + (popSize * XRate);
    P = createPopulation (newpopSize);
    best = createSchedule ();
    best->finish = ABIGNUMBER;

    initPopulation(P, G, edges, ne);

#ifdef SEQUENTIAL
    evaluatePopulation (P, popSize, G, best, PS, 0, tp, bestFile);
#else
    evaluatePopulation (P, popSize, G, best, PS, 0, tp, master_inst);
#endif

    figureReprProbs (P, popSize);

#ifndef SEQUENTIAL
    for (e = 0; e < numEpochs; e++) {
#endif
        /* Run the GA locally for numGens generations */
        for (i = 0; i < numGens; i++) {
            doCrossoverOnPopulation (P, popSize * XRate);
#ifdef SEQUENTIAL
            evaluatePopulation (P, newpopSize, G, best, PS, i, tp, bestFile);
#else
            evaluatePopulation (P, newpopSize, G, best, PS,
                                ((e * numGens) + i), tp, master_inst);
#endif
            figureReprProbs (P, newpopSize);

            /* select new subpopulation */
            P = pickWorkingPopulation (P);
            printPopulationTimes (P, i, popSize);

            mutatePopulation (P, EOMutRate, PAMutRate, G);
        }
        printPopulationTimes (P, i, popSize);

#ifndef SEQUENTIAL
        /* Create a subpopulation to exchange with neighbors */
        /* Send subpopulation */
        sendSubPop(pickSubPop(P), neighbor_list, ngrp);

        EP = createPopulation(ngrp * subpopSize);

        /* receive migrant populations from neighbors */
        rcvSubPop(EP, ngrp, e);

        /* Select a new working population */
        P = newworkPopulation(P, EP, ngrp * subpopSize);

        deletePopulation(EP, ngrp * subpopSize);
        /* end of Epoch */
    }

    /* notify master of termination */
    initsend();
    snd("pgamaster", master_inst, SLAVE_DONE);

    /* Exit PVM */
    leave();
#endif
}
```