Parallel Genetic Algorithms: A Typical MPI Application

Muhammad Alkarouri

February 14, 2002

1 Introduction

Many problems in physics and engineerin can be presented as optimization problems. While in some cases you only need local extrema, in many cases you need global extrema. One of the powerful techniques for global optimization is *Genetic Algorithms*. Here we present an implementation of a genetic algorithm to solve a specific problem, mainly as an exercise in parallel programming.

2 The problem:TSP

Travelling Sales Person, usually abbreviated as TSP, is a widely known problem. The problem is that, given a map of some cities, a travelling person should go a trip in which he passes all the cities and return to the starting point, walking the minimum distance. Features:

- NP-complete
- Unsolvable using calculus
- benchmark for optimization techniques
- targets maximizing a fitness function which is not necessarily differentiable or even continuous

The fitness function of the TSP problem is usually expressed as:

$$Fitness = \sum_{i=0}^{N} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$$

where x_i and y_i are coordinates of city i and $x_N = x_0$, $y_N = y_0$.

3 Genetic Algorithms

This is a technique based on essence on probablistic inspection of the problem space, with some heuristics based on the principle of "surival of the fittest". Features:

- searches *populations* instead of minimizing a single variable
- can be stopped at any point to have a less than optimal solution
- can be thought of as self-correcting {a merit and a drawback}
- easily parallelizable

It depends on some *genetic operators*:

Selection : The best *chromosomes* get better chance to go to the next generation.

Crossover : Crossover of chromosomes gives you new chromosomes of different characteristics. Selection keeps the best.

Mutation : Sometimes a child has a feature not existing in the parents. This will usually enhance the search of the problem space and avoids sstucking in a local extremum.

Elitism : Just keeping the best ones to survive from one generation to the other. This is optional.

Sometimes genetic algorithms are coupled with local search to get *memetic algorithms*. This is not the case here,however.

4 Parallel GA

There is more than one way to parallelize a genetic algorithm program.

Distributed population model in which there is one population distributed among the processes. Can be used in shared memory processing.

Multiple population model is more applicable in message passing environments. Here every process has a different population and every now and them processes exchange parts of their population.

5 The analysis

A program was written using MPI to implement a solution to the TSP problem using the multiple population model. Some points:

- Written from the start as parallel, rather than parallelizing a serial version.
- Does not use an existing library
- Developed step by step, utilizing the self-correcting feature.

5.1 Coding of the TSP path

The path was simply given as an array of cities. This makes special crossover and mutation operators

5.2 crossover

The *permutation* crossover operator was used. Here you get a part of the first chromosome and complete it from the second. For example:

city 1: `0 2 | 1 3 4`
city 2: `2 1 | 3 0 4`
will be crossed over to get:
city 1: `0 2 | 1 3 4`
city 2: `2 1 | 0 3 4`

5.3 mutation

In permutation encoding, mutation is usually done as the exchange of two cities so

```
city: 0 2 3 1 4
        *     *
city: 0 4 3 1 2
```

5.4 elitism

Elitism is the action of keeping the best chromosome(s) on the population without subjecting them to mutation or other operators.

6 MPI programming

In the process of writing the program, the following points were observed

6.1 Debugging

- As this program makes heavy use of pointers, some errors were made (and corrected!). The implementation of MPICH library on Linux does not tolerate this, so in most of the time you get an error immediately.
- Using TotalView is very helpful. I don't know of an open source counterpart.

6.2 Optimizing code

- While possible opportunities to make the code more optimized, actual optimization of code should be left till the end.

- In the first stages make free use of collective communications, while optimizing the serial code.
- In my application, I have found it advantageous to make the server work as a client also. Besides sharing work, this simplifies communications.
- Try to group the communications using MPI_Pack/Unpack or derived types, before overlapping communications with computations
- Only then, use asynchronous communication.
- Always use an instrumenting program to optimize. Don't trust common sense.
- In the MPICH distribution, there is a library called MPE that can be used for instrumentation.

7 Comments

- Genetic algorithms usually get to the global optimal area fairly quick, while they become slower as they approach the solution. They are frequently coupled with other optimization techniques to get better solutions.
- Genetic algorithms usually need time to get the solution that increases linearly with the dimension of the problem.
- Parallel genetic algorithms are scalable as the speed up gained is proportional to the number of processors
- There is a number of generic parallel genetic algorithm libraries available. An example is PGAPACK from Argonne State University.

8 Web Links

- Traveling Salesman Problem - Home Page: http://www.math.princeton.edu/tsp/ is a good resource to know about the TSP problem.
- Introduction to genetic algorithms with Java applets: http://cs.felk.cvut.cz/ xobitko/ga/ . An excellent introduction to genetic algorithms.
- Comparison of various approaches to solving Traveling Salesman Problem: http://www.lips.utexas.edu/ scott/ta/project9/TSPReport.htm . Compares various methods for global optimization.
- The Users Guide to the PGAPack Parallel Genetic Algorithm Library can be downloaded from

  ftp://ftp.cs.bham.ac.uk/pub/Mirrors/ftp.de.uu.net/EC/GA/docs/pgapack-guide.ps.gz

A The program

This program is in the public domain. It is provided "as is".

```
#include <stdio.h>
#include <mpi.h>
#include <math.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include <memory.h>
#define CITYNO 8
#define DATAFILE "cities.txt"


typedef struct scity {
    float longitude;
    float latitude;
} city;
MPI_Datatype MPI_City;

typedef int path[CITYNO];

city cities[CITYNO];

void server(void);
void client(void);
int rank, size;

int main(int argc, char **argv)
{
    int i;
    int err;
    city acity;
    int lenc[2];
    MPI_Aint locc[2];
    MPI_Datatype typc[2];
    MPI_Aint baseaddr;
    // Initialize MPI
    err = MPI_Init(&argc, &argv);
    err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    err = MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Make the city MPI type
    MPI_Address(&acity, &baseaddr);
    lenc[0] = 1;
    MPI_Address(&acity.longitude, &locc[0]);
```

```
    locc[0] -= baseaddr;
    typc[0] = MPI_FLOAT;
    lenc[1] = 1;
    MPI_Address(&acity.latitude, &locc[1]);
    locc[1] -= baseaddr;
    typc[1] = MPI_FLOAT;
    MPI_Type_struct(2, lenc, locc, typc, &MPI_City);
    MPI_Type_commit(&MPI_City);

    if (rank == 0) {
server();
    } else {
client();
    }
    // Finish up
    err = MPI_Finalize();
    return 0;
}

void server(void)
{
    int i;
    FILE *fcities;
    if ((fcities = fopen(DATAFILE, "r")) == 0) {
perror("cities file should be created");
exit(1);
    }
    for (i = 0; i < CITYNO; i++) {
fscanf(fcities, "%f %f", &cities[i].longitude,
       &cities[i].latitude);
    }
    fclose(fcities);
    // with nothing else to do, the server can work as a client
    client();
}

void random_shuffle(path p);
float fit(path p);
int comparepaths(const void *p1, const void *p2);
void crossover(path p1, path p2);
void docrossover(path p1, path p2, path np1, path np2);
void printpath(path p);
void mutate(path p);
void select_crossover_choices(int *p, float *fits);
void random_crossover_choices(int *p, float *fits);
```

```
#define POPULATION_SIZE 1000
#define GENERATIONS 25
#define MUTATIONFACTOR 0.04
#define EXCHANGEPERIOD 10

void client(void)
{
    int i, j;
    path *paths, *newpaths, *tmp;
    float *pathfits;
    struct {
float value;
int index;
    } in, out;
    struct timeval tv;
    int crossover_choices[2 * (POPULATION_SIZE - 2)];
    MPI_Status status;
    paths = (path *) malloc(POPULATION_SIZE * sizeof(path));
    newpaths = (path *) malloc(POPULATION_SIZE * sizeof(path));
    pathfits = (float *) malloc(POPULATION_SIZE * sizeof(float));
    MPI_Bcast(cities, CITYNO, MPI_City, 0, MPI_COMM_WORLD);
    // Initialize random number generator
    gettimeofday(&tv, NULL);
    srandom(tv.tv_usec);
    // initialize old paths
    for (i = 0; i < CITYNO; i++)
paths[0][i] = i;
    for (i = 1; i < POPULATION_SIZE; i++) {
memcpy(paths[i], paths[i - 1], sizeof(paths[i]));
random_shuffle(paths[i]);
    }
    for (j = 0;; j++) {
// sort paths
qsort(*paths, POPULATION_SIZE, sizeof(paths[0]), comparepaths);
if (j == GENERATIONS)
    break;
//exchange
if (j % EXCHANGEPERIOD == 0)
    MPI_Sendrecv(paths[POPULATION_SIZE - 1], CITYNO, MPI_INT,
 (rank + 1) % size, 0, paths[0], CITYNO, MPI_INT,
 (rank - 1) % size, 0, MPI_COMM_WORLD, &status);
// you should use the selection operator
for (i = 0; i < POPULATION_SIZE; i++)
    pathfits[i] = fit(paths[i]);
select_crossover_choices(crossover_choices, pathfits);
// make newpaths
```

```
//elitism
memcpy(newpaths[POPULATION_SIZE - 1],
       paths[POPULATION_SIZE - 1], sizeof(path));
for (i = 0; i < POPULATION_SIZE - 2; i++) {
    docrossover(paths[crossover_choices[2 * i]],
paths[crossover_choices[2 * i + 1]],
newpaths[i], newpaths[i + 1]);
}
for (i = 0; i < POPULATION_SIZE - 2; i++)
    if (random() / RAND_MAX < MUTATIONFACTOR)
mutate(newpaths[i]);
// newpaths finished
random_crossover_choices(crossover_choices, pathfits);
for (i = 0; i < POPULATION_SIZE - 2; i += 2)
    crossover(paths[crossover_choices[i]],
      paths[crossover_choices[i + 1]]);
for (i = 0; i < POPULATION_SIZE - 2; i++)
    if (random() % 100 < 4)
mutate(paths[i]);
    }
    in.index = rank;
    in.value = fit(paths[POPULATION_SIZE - 1]);
    MPI_Allreduce(&in, &out, 1, MPI_FLOAT_INT, MPI_MAXLOC, MPI_COMM_WORLD);
    if (rank == out.index) {
printf("Best fit is:");
printpath(paths[POPULATION_SIZE - 1]);
    }
    free(pathfits);
    free(newpaths);
    free(paths);
}

float sq(float x)
{
    return x * x;
}

float dist(city c1, city c2)
{
    return sqrt(sq(c1.longitude - c2.longitude) +
sq(c1.latitude - c2.latitude));
}

/* crossover of two paths */
void crossover(path p1, path p2)
{
```

```
    int cp, i, j, k;
    path newp1, newp2;
    cp = random() % CITYNO;
    for (i = 0; i < cp; i++)
newp1[i] = p1[i];
    //start p2 from the beginning
    // if an element does not exist add it
    j = 0;
    while (i < CITYNO) {
for (k = 0; k < i; k++)
    if (p2[j] == newp1[k])
break;
if (k == i)
    newp1[i++] = p2[j];
j++;
    }
    for (i = 0; i < cp; i++)
newp2[i] = p2[i];
    //start p1 from the beginning
    // if an element does not exist add it
    j = 0;
    while (i < CITYNO) {
for (k = 0; k < i; k++)
    if (p1[j] == newp2[k])
break;
if (k == i)
    newp2[i++] = p1[j];
j++;
    }
    memcpy(p1, newp1, sizeof(newp1));
    memcpy(p2, newp2, sizeof(newp2));
}

/* mutation of a permutation */
void mutate(path p)
{
    int c1, c2;
    int c;
    c1 = random() % CITYNO;
    c2 = random() % CITYNO;
    c = p[c1];
    p[c1] = p[c2];
    p[c2] = c;
}

/* a function to shuffle cities in a path
```

```
 * it exchanges two random cities for random times
 */
void random_shuffle(path p)
{
    int i;
    for (i = 0; i < random() % CITYNO; i++)
mutate(p);
}

/* a function to evaluate the fitness of the path
 * it will be the -ve of the sum of distances
 * to be maximized for the minumum distance
 */
float fit(path p)
{
    int i;
    float sum = 0;
    for (i = 0; i < CITYNO; i++)
sum += dist(cities[p[i]], cities[p[(i + 1) % CITYNO]]);
    return sum;
}

int comparepaths(const void *p1, const void *p2)
{
    float f1, f2;
    f1 = fit(*(path *) p1);
    f2 = fit(*(path *) p2);
    if (f1 < f2)
return 1;
    else
return (f1 == f2) ? 0 : -1;
}

void printpath(path p)
{
    int i, f;
    for (i = 0, f = 1; i < CITYNO; i++)
f *= (p[i] == 0) ? 1 : p[i];
    for (i = 0; i < CITYNO; i++)
printf("%d\t", p[i]);
    printf("fit=%f\n", fit(p));
}

void random_crossover_choices(int *p, float *fits)
{
    int i, j;
```

```
    for (i = 0; i < POPULATION_SIZE - 2; i++)
p[i] = i;
    for (i = 0; i < random() % (POPULATION_SIZE - 2); i++) {
int c1, c2;
int c;
c1 = random() % (POPULATION_SIZE - 2);
c2 = random() % (POPULATION_SIZE - 2);
c = p[c1];
p[c1] = p[c2];
p[c2] = c;
    }
}

void select_crossover_choices(int *p, float *fits)
{
    int i, j;
    float tmp, sum = 0;
    tmp = fits[POPULATION_SIZE - 1];
    //implements the selection operator
    //pre: fits array is sorted largest first. less is better
    for (i = 0; i < POPULATION_SIZE; i++) {
fits[i] = tmp - fits[i];
sum += fits[i];
    }
    for (i = 0; i < POPULATION_SIZE; i++) {
fits[i] /= sum;
    }
    for (i = POPULATION_SIZE - 2; i >= 0; i--) {
fits[i] += fits[i + 1];
    }
    for (i = 0; i < 2 * (POPULATION_SIZE - 2); i++) {
//get candidates for crossover
float r = random() / RAND_MAX;
for (j = 0; j < POPULATION_SIZE; j++)
    if (fits[j] < r)
break;
p[i] = j - 1;
    }
}

/* crossover of two paths */
void docrossover(path p1, path p2, path np1, path np2)
{
    int cp, i, j, k;
    cp = random() % CITYNO;
    for (i = 0; i < cp; i++)
```

```
np1[i] = p1[i];
    //start p2 from the beginning
    // if an element does not exist add it
    j = 0;
    while (i < CITYNO) {
for (k = 0; k < i; k++)
    if (p2[j] == np1[k])
break;
if (k == i)
    np1[i++] = p2[j];
j++;
    }
    for (i = 0; i < cp; i++)
np2[i] = p2[i];
    //start p1 from the beginning
    // if an element does not exist add it
    j = 0;
    while (i < CITYNO) {
for (k = 0; k < i; k++)
    if (p1[j] == np2[k])
break;
if (k == i)
    np2[i++] = p1[j];
j++;
    }
}
```