Multi-start PSO with CCD Local Optimizer (MPSO_CCD)

A Global Optimization Technique

User Manual

by

Dr. Hum Nath Bhandari, Dr. Philip W. Smith

Email: humnath.bh@gmail.com, philip.smith@ttu.edu

Table of Contents

1. Serial Implementation 1
   1.1 Flowchart 1
   1.2 Required Files 2
      1.2.1 Header Files 2
      1.2.2 C Files 2
   1.3 Compiling and Executing the Code 3
      1.3.1 Compiling the Code 3
      1.3.2 Executing the Code 3
         1.3.2.1 Using Command Line 4
         1.3.2.2 Using Job Script 4
   1.4 Example 4
      1.4.1 `functions.h` file 5
      1.4.2 `functions.c` file 5
      1.4.3 `main.c` file 6
      1.4.4 Outputs 7
   1.5 Optimization Algorithms 9
      1.5.1 PSO Algorithm 9
         1.5.1.1 Introduction 9
         1.5.1.2 Implementation of the PSO Algorithm 10
      1.5.2 CCD Algorithm 11
         1.5.2.1 Introduction 11
         1.5.2.2 Implementation of the CCD Algorithm 11
      1.5.3 Brent Algorithm 12
         1.5.3.1 Introduction 12
         1.5.3.2 Implementation of the Brent Algorithm 13
      1.5.4 MPSO_CCD Algorithm 13
         1.5.4.1 Introduction 13
         1.5.4.2 Implementation of the MPSO_CCD Algorithm 14

1.6 Supportive Functions 16
1.6.1 Supportive Functions for PSO 16
1.6.2 Math Related Functions 16
1.6.3 Random Number Generators 17
2. Parallel Implementation 18
2.1 Required Files 18
2.1.1 Header Files 18
2.1.2 C Files 18
2.2 Compiling and Executing the Parallel Code 19
2.2.1 Compiling the Parallel Code 19
2.2.2 Executing the Parallel Code 20
2.2.2.1 Using Command Line 20
2.2.2.2 Using Job Script 20
2.3 Parallel PSO Versions 20
2.4 Example of Parallel Implementation 21
2.4.1 `mpimain.c` file 21
2.5 Source Code 24
Bibliography 25

Abstract

This manual provides the stepwise guidelines for the serial as well as parallel implementation of MPSO_CCD algorithm [1]. It is a global optimization technique which combines the particle swarm optimization (PSO) algorithm with the cyclic coordinate descent(CCD) local optimizer. Therefore it incorporates the global search characteristics of the PSO algorithm with the local search capability of the CCD algorithm. This unique combination provides a better performance by balancing the exploration-exploitation trade-off. We further implement the MPSO_CCD algorithm in a parallel environment to increase the scalability.

Chapter 1

Serial Implementation

This chapter presents the compilation and execution of the serial version of the MPSO_CCD algorithm written in C programming language.

1.1 Flowchart

The implementation procedures can be demonstrated by the following flowchart.

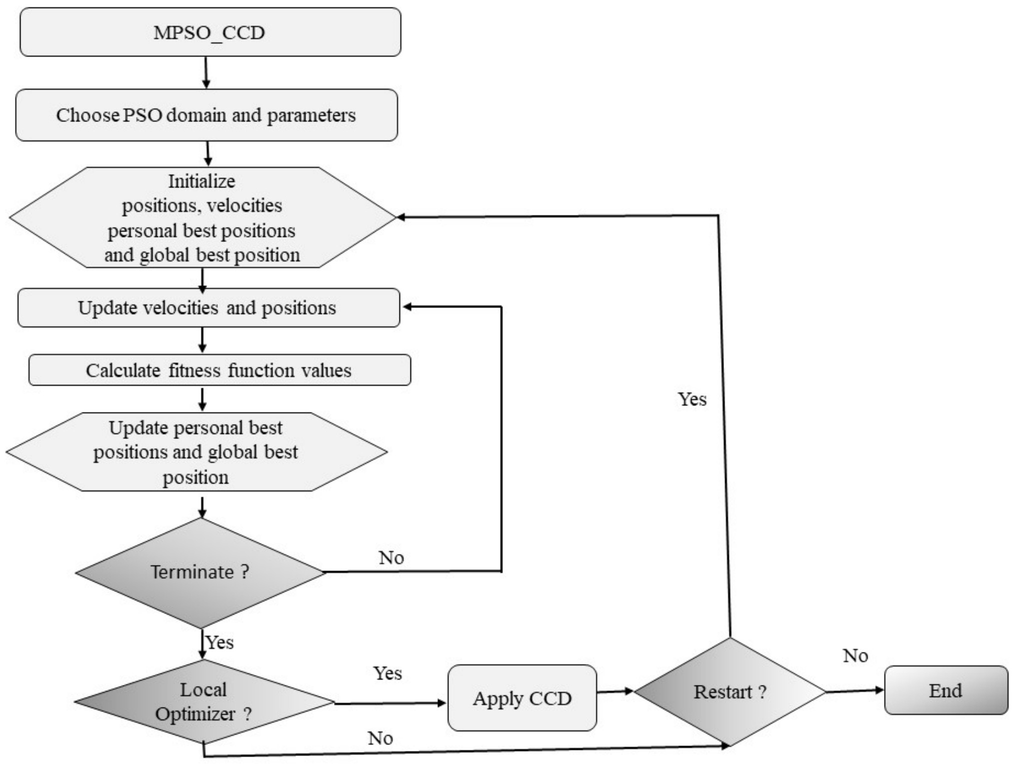
MPSO_CCD
Choose PSO domain and parameters
Initialize positions, velocities personal best positions and global best position
Update velocities and positions
Calculate fitness function values
Update personal best positions and global best position
Terminate ?
No
Yes
Local Optimizer ?
Yes
No
Apply CCD
Restart ?
Yes
No
End

Figure 1.1. Flowchart of the MPSO_CCD algorithm.

1.2 Required Files

The following are the files required to implement the serial version of the PSO algorithm. All files must be in the same directory, say "mpso_ccd_serial".

1.2.1 Header Files

1. Basic C Libraries.

   `stdio.h`, `stdlib.h`, `string.h`, `math.h`, `time.h`, `assert.h`

2. Program Related Header files.

   `pso.h   functions.h   randomlib.h  matmul2.h`

1.2.2 C Files

1. `pso.c`: PSO related functions file.

   This file contains the function headers of optimization algorithms such as PSO algorithm, CCD algorithm, Brent algorithm, MPSO_CCD, and some other supportive functions. The corresponding function headers are included in `pso.h` file.

2. `functions.c` : Objective functions file.

   This file contains the objective functions. In the current form, it has some of the common benchmark functions such as Sphere, Rastrigin, Rosenbrock, Griewank, Ackley, etc. If required, users can define their own functions and put in this file. The function, called `functionLookup()`, is then used to select a particular objective function to be minimized.

3. `matmul2.c`: Math related functions file.

   This file consists some math related functions which are not included in the basic math library, called `math.h`, but they are essential to facilitate the PSO algorithm implementation such as matrix multiplications, transposing matrix, etc. The corresponding function headers are included in the file `matmul2.h`.

4. `randomlib.c`: Random numbers generators.

This file contains the functions related to random number generators. These random number generators are based on the algorithm in a FORTRAN version [3]. Since the PSO algorithm is stochastic algorithm, these functions are essential to initialize and re-initialize the PSO particle positions, velocities, etc.

5. `main.c`: Main file (Driver file)

   This file is a driver of the entire program and contains the `main()` function. It performs the following tasks:

   - Creates necessary variables.
   - Initializes random number generators with seeds.
   - Initializes PSO domain.
   - Executes optimization function: `mpso_ccd()`.
   - Writes outputs.

6. `job.sh`: Optional file

1.3 Compiling and Executing the Code

Assuming all above mentioned files are in the same directory, the following commands are used to compile and run the program.

1.3.1 Compiling the Code

The following command can be used for compiling the code.

```
$g++ main.c pso.c functions.c matmul2.c randomlib.c -o pso_opt
```

It produces `pso_opt` as a binary object.

1.3.2 Executing the Code

The following command can be used for executing the code.

1.3.2.1 Using Command Line

1. Linux/Mac Machines

   ```
   $./pso_opt
   ```

2. Windows Machines

   ```
   $pso_opt
   ```

1.3.2.2 Using Job Script

If the user chooses to run the program using job script submission, the following commands should be included in the `job.sh` file.

```
#!/bin/bash
#$ -V
#$ -cwd
#$ -j y
#$ -S /bin/bash
#$ -N pso
#$ -q serial
#$ -P hostname
$./pso_opt
OR
$pso_opt
```

The job should be submitted using the following command.

```
qsub job.sh
```

We implemented the serial code in the hrothgar cluster of high performance computing center (HPCC) at Texas Tech University. The job file might be slightly different depending on the other systems and clusters.

1.4 Example

We provide an example `main.c` file which is used to minimize the Rosenbrock function. The Rosenbrock function can be defined as follows.

$$f(\mathbf{x}) = \sum_{j=1}^{n-1}[100(x_{j+1} - x_j^2) + (x_j - 1)^2]$$

It has a global minimum value $f(\mathbf{x}^*) = 0$ at $\mathbf{x}^* = (1, 1, 1, ..., 1)$.

1.4.1 `functions.h` file

```
double Rosenbrock(double * x, int nd);
```

1.4.2 `functions.c` file

```
double Rosenbrock(double * x, int nd) {
    double sum = 0;
    int i;
    for(i=0;i<(nd-1);i++){
       sum += 100*pow(x[i+1]- pow(x[i],2),2)+pow(x[i]-1,2);
    }
 return(sum);
}
```

1.4.3 **main.c** file

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "randomlib.h"
#include "matmul2.h"
#include "pso.h"
#include "assert.h"
#include "functions.h"
int main(int argc,char*argv[]){

        int i,j,k, seed1,seed2;
        srand((unsigned)time(NULL));
        seed1= rand()%30000; seed2= rand()%30000;
        RandomInitialise(seed1,seed2);

        char funName[80];
        strcpy(funName, "Rosenbrock");
        fptr f  = functionLookup(funName);
        int np =  500, nd = 5, ni = 500, maxRun = 50;
        double l  = -10, u = 10;
    /*
                nd:  dimension of the problem to be optimized
                np:  number of  PSO particles
                ni:  maximum iterations allowed per run
                maxRun: maximum runs (re-starts) allowed
                l: lower bound of each component of the domain
                u: upper bound of each component of the domain
        */
        double*lb = (double*)malloc(sizeof(double)*nd);
        double*ub = (double*)malloc(sizeof(double)*nd);
        double *gbest =  (double*)malloc(sizeof(double)*nd);
        for(i = 0; i < nd; i++){
                        lb[i] = l; ub[i] = u;
                        gbest[i] = (ub[i]-lb[i])/3;
                }
        double start,finish, ellapse_time;
        double value = 9999999;

        /*=== Start optimization ===*/
        start = time(NULL);

        mpso_ccd(f, nd, np, ni, maxRun, lb, ub, &value, gbest);

        finish= time(NULL);

        ellapse_time = difftime(finish, start);

        /* === Writing outputs ====*/
        printf("\n ==== PSO Parameters Used ====\n");
        printf("Best value obtained: %lf\n",value);
        printf("Random seeds: %d, %d\n",seed1,seed2);
        printf("Dimension of problem: %d\n",nd);
        printf("Number of PSO particles used: %d\n",np);
        printf("Maximum iterations per run allowed: %d\n",ni);
        printf("Maximum runs allowed : %d\n", maxRun);
        printf("Wall-clock time (Seconds): %lf\n",ellapse_time);
        printf("\n === Optimal Solution ====\n");
        for(j=0;j<nd;j++){printf("%lf\n", gbest[j]);}
        free(lb);free(ub);free(gbest);

        return 0;
}
```

1.4.4 Outputs

```
==== PSO Parameters Used ====
Best value obtained: 0.000000
Random seeds: 28259, 25033
Dimension of problem: 5
Number of PSO particles used: 500
Maximum iterations per run allowed: 500
Maximum runs allowed : 50
Wall-clock time (Seconds): 3.000000

=== Optimal Solution ====
1.000000
1.000000
1.000000
1.000000
1.000000
```

1.5 Optimization Algorithms

This section provides a brief overview of the optimization algorithms that are used in this program.

1.5.1 PSO Algorithm

1.5.1.1 Introduction

The particle swarm optimization algorithm (PSO) is a bio-inspired stochastic optimization technique developed by [2]. It is a population-based search algorithm which mimics the behavior of bird flocking, fish schooling, etc. In the algorithm, the total population is called swarm, and individuals are called particles. Particles are points in the search space of the underlying optimization problem. Consider the PSO system with N_p particles and n dimensions. Then the i^{th} particle maintains a triple of vectors $(\mathbf{X}_i^{(k)}, \mathbf{V}_i^{(k)}, \mathbf{Y}_i^{(k)}) \in \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n$ for $1 \leq i \leq N_p$, where $\mathbf{X}_i^{(k)} \in \mathbb{R}^n$ is the current position, $\mathbf{V}_i^{(k)} \in \mathbb{R}^n$ is the current velocity, and $\mathbf{Y}_i^{(k)} \in \mathbb{R}^n$ is the personal best position. For simplicity, the dimension of vectors is not written explicitly, which is the same as the dimension of the underlying optimization problem. The velocity and the position of each particle i are updated by using the following equations[8].

$$\mathbf{V}_i^{(k+1)} = w\mathbf{V}_i^{(k)} + c_1 r_1(\mathbf{Y}_i^{(k)} - \mathbf{X}_i^{(k)}) + c_2 r_2(\hat{\mathbf{Y}}^{(k)} - \mathbf{X}_i^{(k)}) \tag{1.1}$$

$$\mathbf{X}_i^{(k+1)} = \mathbf{X}_i^{(k)} + \mathbf{V}_i^{(k+1)} \tag{1.2}$$

where $\hat{\mathbf{Y}}^{(k)}$ represents the global best position and $r_1, r_2 \in U[0,1]$ are random numbers chosen from a random uniform distribution $U[0,1]$. There are three components in the velocity equation with corresponding learning parameters $w, c_1, c_2 \in \mathbb{R}_0^+$ where $\mathbb{R}_0^+$ is the set of positive real numbers including zero. Here w is a weight parameter for the inertia component $w\mathbf{V}_i^{(k)}$, c_1 is a parameter for the cognitive component $c_1 r_1(\mathbf{Y}_i^{(k)} - \mathbf{X}_i^{(k)})$, and c_2 is a parameter for the social component $c_2 r_2(\hat{\mathbf{Y}}^{(k)} - \mathbf{X}_i^{(k)})$. Parameters c_1 and c_2 are also called acceleration coefficients.

Furthermore, the personal best and the global best positions are updated as follows:

$$\mathbf{Y}_i^{(k+1)} = \left\{ \begin{array}{c} \mathbf{X}_i^{(k+1)} \text{ if } f(\mathbf{X}_i^{(k+1)}) < f(\mathbf{Y}_i^{(k)}) \\ \mathbf{Y}_i^{(k)} \text{ otherwise} \end{array} \right\} \tag{1.3}$$

$$\hat{\mathbf{Y}}^{(k+1)} = \arg\min\{f(\mathbf{Y}_1^{(k+1)}), f(\mathbf{Y}_2^{(k+1)}), ..., f(\mathbf{Y}_{N_p}^{(k+1)})\} \tag{1.4}$$

The PSO algorithm has been successfully applied to many optimization problems such as pattern recognition, clustering, classification, neural network training, sensor networks, scheduling, robotics, signal processing, and power systems [4]. Because of the easy implementation and fast convergence to acceptable solutions, the PSO algorithm has received popularity in recent years [4].

1.5.1.2 Implementation of the PSO Algorithm

Name: **`pso()`**

Purpose: to minimize a multi-dimensional optimization problem using the standard PSO algorithm.

Usage: `void pso(double (*fn)(double* x, int nd), int nd, int np, int ni, double* lb, double* ub, double* value, double* gbest)`

Arguments:

- `fn` - function to be minimized with the following arguments:
- `nd` - dimension of the problem (Input).
- `np` - number of particles (Input).
- `ni` - maximum number of iterations allowed (Input).
- `lb` - vector of length `nd` which consists lower bounds domain (Input).
- `ub` - vector of length `nd` which consists upper bounds of domain (Input).
- `value` - function value at the `gbest` (Input/Output).
- `gbest` - vector of length `nd` containing the best estimate of the minimum found (Input/Output).

1.5.2 CCD Algorithm

1.5.2.1 Introduction

The CCD algorithm uses the coordinates of $\mathbb{R}^n$ as the search directions and searches along one coordinate at a time while fixing the other coordinates. Starting with an initial guess $\mathbf{x}^{(1)} = (\mathbf{x}_1^{(1)}, \mathbf{x}_2^{(1)}, \mathbf{x}_3^{(1)} ..., \mathbf{x}_n^{(1)})$, the next estimate $\mathbf{x}^{(k+1)}$ is obtained from $\mathbf{x}^{(k)}$ by iteratively solving the following one dimensional optimization problems:

$$\mathbf{x}_j^{k+1} = \arg\min_{t \in \mathbb{R}} f(\mathbf{x}_1^{k+1}, \ldots, \mathbf{x}_{j-1}^{k+1}, t, \mathbf{x}_{j+1}^{k}, \ldots, \mathbf{x}_n^{k})$$

for $j = 1, 2, \ldots, n$. Therefore, a sequence of points $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \ldots$ are generated iteratively so that

$$f(\mathbf{x}^{(1)}) \geq f(\mathbf{x}^{(2)}) \geq f(\mathbf{x}^{(3)}) \geq \ldots$$

The CCD algorithm like steepest descent converges to a stationary point if the objective function f is differentiable. However, for non-differentiable functions, the algorithm can stall at a point which may not be a stationary point, leading to premature termination [5] . One of the reasons for such behavior might be because of the presence of a valley caused by non-differentiability. This issue can be resolved by using a strategy, called *acceleration step* [5].

1.5.2.2 Implementation of the CCD Algorithm

This is a local search algorithm, which is used to refine the solution obtained from the PSO algorithm.

Name: `ccd_local_optimizer()`

Purpose: to minimize multi-dimensional function using cyclic coordinate descent (CCD) algorithm.

Usage: **`void`** `ccd_local_optimizer(`**`double`** `(*fun)(`**`double*`** `y,` **`int`** `nd),` **`int`** `nd,` **`double*`** `lb,` **`double*`** `ub,` **`double*`** `value,` **`double*`** `x`

Arguments:

- `fun` - function to be minimized (Input).
- `nd` - dimension of the problem (Input).
- `lb` - vector of length `nd` which consists lower bounds of domain (Input).
- `ub` - vector of length `nd` which consists upper bounds of domain (Input).
- `value` - function value (Input/Output).
- `x` - initial guess (Input/Output).

1.5.3 Brent Algorithm

1.5.3.1 Introduction

The Brent algorithm [6] combines the bisection method, secant method, and inverse quadratic interpolation and searches for a function minimum over the given range. It was derived from an earlier algorithm by Theodorus Dekker [7]. In our implementation, the original Brent algorithm has been slightly modified, where the golden section method is used instead of the secant method. Therefore, the modified Brent algorithm combines golden section, bisection, and quadratic interpolation methods. It has the ability to switch between gold section and quadratic interpolation during the execution. It converges super-linearly at a rate of about 1.324 to the solution as long as a solution exists within the specified interval. The following are the inputs and outputs of Brent algorithm.

Inputs: The Brent algorithm works in the interval domain $[l, u]$ with lower bound l and upper bound u, $l < u$, so that the minimum will be searched over a range $[l, u]$. The objective function for the algorithm is a univariate function $g(x)$, $x \in [l, u]$. An acceptable tolerance $tol > 0$ is provided as a termination criterion.

Outputs: The algorithm returns an estimate x for the local minimum of g with accuracy: $3\sqrt{\epsilon}\,|x| + tol$ where $\sqrt{\epsilon} = 10^{-6}$. It obtains an estimate of a local minimum of the objective function which coincides with the global minimum only if the function is unimodal. If the function does not have local minimum within the given range, it returns l if $g(l) < g(u)$, otherwise it returns u.

1.5.3.2 Implementation of the Brent Algorithm

This function performs one dimensional line search using the Brent algorithm and is called by `ccd_local_optimizer()`.

Name: `brent_algorithm()`
Purpose: to minimize one dimensional function using a Brent algorithm.

Usage: **`double`** `brent_algorithm(`**`double`** `l,` **`double`** `u,`
`double``(*g)(`**`double`** `t),` **`double`** `tol)`

Arguments:

- `l` - lower bound of the interval $[l, u]$ (Input).
- `u` - upper bound of the interval $[l, u]$ (Input).
- `g` - one dimensional function to be minimized.
- `tol` - Acceptable tolerance for the minimum location.

1.5.4 MPSO_CCD Algorithm

1.5.4.1 Introduction

To improve the performance of the PSO algorithm, a multi-start approach should be used where the standard PSO (i.e `pso()`) is repeated multiple times by re-initializing the particles. After each run, the global best position is recorded and re-used as an initial guess for the next run. The particles are re-initialized in the entire search space and the standard PSO is executed again for the next run. This process is repeated multiple times and has the ability to generate points in each part of the search space, increasing the exploration. This strategy significantly improves the global search capability of the standard PSO algorithm. However, it still lacks the ability to exploit the local domain as compared with some existing local search algorithms. To increase the local search ability, we combine the PSO algorithm with the CCD algorithm to make a new hybrid algorithm, called MPSO_CCD. It increases both exploration as well as exploitation abilities.

1.5.4.2 Implementation of the MPSO_CCD Algorithm

This algorithm calls the `pso()` and `ccd_local_optimizer()` repeatedly. First, the best estimate `gbest` is obtained by using the `pso()` algorithm. Then the `gbest` vector is provided to the `ccd_local_optimizer()` as an initial guess. It further improves the estimate using the `brent_algorithm()`. This process is repeated `maxRun` times and can be summarized as follows.

Step 1: Apply `pso()`

Step 2: Apply `ccd_local_optimizer()`

Step 3: Repeat Step 1 and Step 2 `maxRun` times or until early criterion criterion is reached.

Name: **`mpso_ccd()`**

Purpose: to minimize a multi-dimensional optimization problem.

Usage:
```
void mpso_ccd(double (*fn)(double* x, int nd),
int nd, int np, int ni, int maxRun,
 double* lb, double* ub, double* value, double* gbest)
```

Arguments:

- `fn` - function to be minimized (Input).
- `nd` - dimension of the problem (Input).
- `np` - number of particles (Input).
- `ni` - maximum number of iterations per run is allowed (Input).
- `maxRun` - maximum number of runs is allowed (Input).
- `lb` - vector of length `nd` which consists lower bounds domain (Input).

- `ub` - vector of length `nd` which consists upper bounds of domain (Input).
- `value` - function value at the `gbest` (Input/Output).
- `gbest` - vector of length `nd` containing the best estimate of the minimum found (Input/Output).

1.6 Supportive Functions

1.6.1 Supportive Functions for PSO

Some additional supportive functions are necessary to implement MPSO_CCD algorithm. These functions are supplied to perform some specific tasks such as initializing domain, checking boundary of domain, adjusting domain, etc.

1. `void update_best(double * g, double * x, int nd)`
2. `int lower_bound_check(double g, double l,`
   `double u, double epsilon)`
3. `int upper_bound_check(double g, double l,`
   `double u, double epsilon)`

1.6.2 Math Related Functions

Some math related functions are also necessary to facilitate the PSO algorithm implementation such as matrix multiplications, transposing matrix, etc. The c code of these functions are included in the file `matmul2.c` and corresponding function headers are included in the file `matmul2.h`.

1. `struct matrix { double ** mat; int row; int col };`
2. `struct matrix * get_matrix( int row , int col)`
3. `struct matrix * matrix_mult( struct matrix* m1,`
   `struct matrix* m2)`
4. `struct matrix * transpose( struct matrix * m )`
5. `double ** get_mat( int row , int col)`
6. `void free_matrix( struct matrix * m)`
7. `struct matrix * read_matrix( char* filename,`
   `int nrow, int ncol)`
8. `void  write_matrix( char* filename, struct matrix * m)`

1.6.3 Random Number Generators

The files `randomlib.c` and `randomlib.h` contain some functions related to random number generators. These random number generators are based on the algorithm in a FORTRAN version[3].

1. `void RandomInitialise(int seed1, int seed2)`
2. `double RandomUniform(void)`
3. `double RandomGaussian(double a, double b)`
4. `int RandomInt(int n1, int n2)`
5. `double RandomDouble(double a, double b)`

Chapter 2

Parallel Implementation

2.1 Required Files

The following are the files required to implement the parallel version of the MPSO_CCD algorithm. We call this parallel algorithm as PMPSO_CCD. All files must be in the same directory, say “mpso_ccd_parallel”.

2.1.1 Header Files

1. Basic C Libraries.

   `stdio.h`, `stdlib.h`, `string.h`, `math.h`, `time.h`, `assert.h`

2. MPI Library.

   `mpi.h`

3. Program Related Header files.

   `mpipso.h   functions.h   randomlib.h  matmul2.h`

2.1.2 C Files

1. `mpipso.c`: PSO related functions file.

   This file contains the function headers of optimization algorithms such as parallel PSO, CCD, Brent algorithm, parallel MPSO_CCD algorithm, and some other supportive functions. The corresponding function headers are included in `mpipso.h` file.

2. `functions.c` : Objective functions file.

   This file is the same as that was in the serial case.

3. `matmul2.c`: Math related functions file.

   This file is the same as that was in the serial case.

4. `randomlib.c`: Random numbers generators.

   This file is the same as that was in the serial case.

5. `mpimain.c`: Main file (Driver file)

   This file is a driver of the entire program and contains the `main()` function. It performs the following tasks:

   - Creates necessary variables.
   - Initializes random number generators with seeds.
   - Setups MPI environment.
   - Initializes PSO domain.
   - Executes optimization function: `pmpso_ccd()`.
   - Writes outputs.

6. `mpijob.sh`: Optional file

2.2 Compiling and Executing the Parallel Code

Assuming all above mentioned files are in the same directory, the following commands are used to compile and run the program.

2.2.1 Compiling the Parallel Code

The following command can be used for compiling the parallel code.

Step 1: `$module load intel`
Step 2: `$load impi`
Step 3:

```
$mpicc -o mpipso_opt mpimain.c matmul2.c, randomlib.c mpipso.c
functions.c -lm
```

It produces `mpipso_opt` as a binary object.

2.2.2 Executing the Parallel Code

The following command can be used for executing the code.

2.2.2.1 Using Command Line

```
$mpirun -np NCores ./mpipso_opt
```

2.2.2.2 Using Job Script

If the user chooses to run the program using job script submission, the following commands should be included in the `mpijob.sh` file.

```
#!/bin/bash
#$ -V
#$ -cwd
#$ -j y
#$ -S /bin/bash
#$ -N mpipso
#$ -o $JOB_NAME.o$JOB_ID
#$ -e $JOB_NAME.e$JOB_ID
#$ -q queue type
#$ -pe fill num_processors
#$ -P hostname
module load intel impi
mpirun --machinefile machinefile.$JOB_ID -np $NSLOTS
./mpipso_opt
```

The job should be submitted using the following command.

```
qsub mpijob.sh
```

We implemented the parallel code in the hrothgar cluster of high performance computing center (HPCC) at Texas Tech University. The job file might be slightly different depending on the other systems and clusters.

2.3 Parallel PSO Versions

The PSO algorithm can be executed in parallel by using some additional MPI commands in the existing serial code. The parallel versions of the PSO are very similar to their corresponding serial versions except one additional argument, called `myrank`. This additional argument is used to keep track of processor rank.

1. Parallel PSO: **`mpipso()`**

   Usage: **`void`** `mpipso(`**`double`** `(*fn)(`**`double*`** `x,` **`int`** `nd),` **`int`** `nd,` **`int`** `np,` **`int`** `ni,` **`double*`** `lb,` **`double*`** `ub,` **`double*`** `value,` **`double*`** `gbest`**`int`** `myrank)`

2. Parallel MPSO_CCD: **`pmpso_ccd()`**

   Usage: **`void`** `pmpso_ccd(`**`double`** `(*fn)(`**`double*`** `x,` **`int`** `nd),`

   `int` `nd,` **`int`** `np,` **`int`** `ni,` **`int`** `maxRun,` **`double*`** `lb,` **`double*`** `ub,` **`double*`** `value,` **`double*`** `gbest,`

   `int` `myrank )`

2.4 Example of Parallel Implementation

We provide an example `mpimain.c` file which is used to minimize the Rosenbrock function. The `functions.h` file and `functions.c` file are the same as that were in the serial case.

2.4.1 `mpimain.c` file

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"
#include <math.h>
#include <time.h>
#include "randomlib.h"
#include "matmul2.h"
#include "pso.h"
#include "assert.h"
```

```
#include "functions.h"

int main(int argc,char*argv[]){

        int i,j,k, seed1,seed2;
        srand((unsigned)time(NULL));
        seed1= rand()%30000; seed2= rand()%30000;

        char funName[80];
        strcpy(funName, "Rosenbrock");
        fptr f  = functionLookup(funName);

        int np = 500, nd = 5, ni = 500, maxRun = 50;
        double l  = -10, u = 10;
        /*
        np:  number of  PSO particles
        nd:  dimension of the problem to be optimized
        ni:  maximum iterations allowed per run
        maxRun: maximum runs (re-starts) allowed
        l: lower bound of each component of the domain
        u: upper bound of each component of the domain
        */

        double*lb = (double*)malloc(sizeof(double)*nd);
        double*ub = (double*)malloc(sizeof(double)*nd);
        double *gbest =  (double*)malloc(sizeof(double)*nd);
        for(i = 0; i < nd; i++){
                lb[i] = l; ub[i] = u;
                gbest[i] = (ub[i]-lb[i])/3;
         }

        /*==== Setting up MPI environment ====*/
        int myrank, comsize;
        MPI_Init(&argc, &argv);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &comsize);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Barrier(MPI_COMM_WORLD);

RandomInitialise(myrank + seed1, seed2);

double start,finish, ellapse_time;
double value = 9999999;

/*=== Start optimization ===*/
start = MPI_Wtime(NULL);

pmpso_ccd(f, nd, np, ni, maxRun, lb, ub, &value, gbest,myrank);

finish = MPI_Wtime(NULL);

ellapse_time = difftime(finish, start);

/* === Writing outputs ====*/
if(myrank == 0){
printf("\n ==== PSO Parameters Used ====\n");
printf("Best value obtained: %lf\n",value);
printf("Random seeds: %d, %d\n",seed1,seed2);
printf("Number of processors used: %d\n",comsize);
printf("Dimension of problem: %d\n",nd);
printf("Number of PSO particles used: %d\n",np);
printf("Maximum iterations per run allowed: %d\n",ni);
printf("Maximum runs allowed : %d\n", maxRun);
printf("Wall-clock time (Seconds): %lf\n",ellapse_time);

printf("\n**========== Optimal Solution ======**\n");
for(j=0;j<nd;j++){printf("%lf\n", gbest[j]);}
}
MPI_Finalize();
```

```
        free(lb); free(ub); free(gbest);
        return 0;
}
```

2.5 Source Code

The complete source code of both serial and parallel implementation can be found on GitHub: `https://github.com/humnath5/MPSO_CCD_Method`.

Bibliography

[1] Bhandari, H. N., Ma, X., Paul, A. K., Smith, P., & Hase, W. L. (2018). PSO Method for Fitting Analytic Potential Energy Functions. Application to I(H2O). Journal of chemical theory and computation, 14(3), 1321-1332.

[2] Kennedy, J. and Eberhart R. (1995). Particle swarm optimization. Proc. of IEEE Intl. Conference on Neural Networks, Perth, Australia, November 27-December 1, 4:19421948.

[3] Marsaglia, G., Zaman, A., & Tsang, W. W. (1990). Toward a universal random number generator. Stat. Prob. Lett., 9(1), 35-39.

[4] Poli, R. (2008). Analysis of the publications on the applications of particle swarm optimisation. Journal of Artificial Evolution and Applications.

[5] Bazarra, M., & Shetty, C. (1979). Nonlinear programming, theory and algorithms. John Wiley and Sons.

[6] Brent, R. (1973). Algorithms for minimization without drivatives. Prentice-Hall.

[7] Dekker, T. (1969). Finding a zero by means of successive linear interpolation. Constructive aspects of the fundamental theorem of algebra, 3751.

[8] Van den Bergh, F. (2002). An analysis of particle swarm optimization. November, Ph. D. Dissertation, Faculty of Natural and Agricultural Sci., Univ. Petoria, Pretoria, South Africa.