Evolução de Instruções com Programação Genética Linear Paralela via MPI

Elivelton Botelho Pinheiro[1] **, Mateus Santana Gutemberg**[1]

[1]Universidade Federal do Maranhão (UFMA) – CCET
Departamento de Informática
São Luís – MA – Brasil

{elivelton.botelho, mateus.gutemberg}@discente.ufma.br

Resumo. *Este relatório descreve o desenvolvimento incremental de um sistema de Programação Genética Linear (PGL) com avaliação assíncrona paralela usando MPI, organizado em quatro etapas conforme o roteiro da disciplina.*

Palavras-chave: Programação Genética Linear, MPI, Computação Paralela, Avaliação Assíncrona, Otimização Evolutiva

1. Introdução

A disciplina de Computação Paralela propôs o desenvolvimento de um programa de Programação Genética Linear (PGL) em quatro entregas incrementais, descritas no documento oficial do projeto. Este relatório consolida a experiência adquirida, decisões de design e resultados num único documento.

2. Etapa 1: Cromossomos e Função de Aptidão

2.1. Especificação da Linguagem Binária

Cada cromossomo é formado por um vetor inteiro de tamanho 16(representando os 16 bits), no entanto o tamanho pode aumentar dependendo da expressão de benchmark a ser calculada. 4 bits de `opcode` e 4 bits de operando. Os quatro registradores (`R0-R3`) são de 16 bits; `R3` armazena o fitness.

3. Exemplo Básico em MPI

O código a seguir ilustra o uso mínimo da biblioteca MPI: cada processo descobre seu *rank*, o número total de processos e imprime uma mensagem.

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);            /* inicializa o MPI */

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* id do processo
       ↪ */
```

```
10      MPI_Comm_size(MPI_COMM_WORLD, &size); /* n   total de
            ↪ processos */
11
12      printf("Ol ␣do␣processo␣%d␣de␣%d!\n", rank, size);
13
14      MPI_Finalize();                     /* finaliza o MPI */
15      return 0;
16  }
```

Listing 1. Hello World paralelo com MPI

Compilação e execução.

```
$ mpicc hello_mpi.c -o hello_mpi
$ mpirun -np 4 ./hello_mpi
Olá do processo 0 de 4!
Olá do processo 1 de 4!
Olá do processo 2 de 4!
Olá do processo 3 de 4!
```

4. Funções Fundamentais

Nesta seção listamos, em ordem lógica, todas as funções implementadas no arquivo `library.c`, seguidas de comentários que destacam seus papéis, premissas e eventuais limitações. Além disso, também há todas as constantes utilizadas ao longo do programa.

4.1. Instruções Aritmético-Lógicas

4.1.1. `add`

```
1  int add(int regA, int regB){
2      return regA + regB;
3  }
```

Listing 2. Soma entre registradores

Soma valores de dois registradores e devolve o resultado. Implementação direta em uma linha.

4.1.2. `sub`

```
1  int sub(int regA, int regB){
2      return regA - regB;
3  }
```

Listing 3. Subtração

Subtrai `regB` de `regA`.

4.1.3. `mult`

```
int mult(int regA, int regB){
    return regA * regB;
}
```

Listing 4. Multiplicação

Multiplicação simples.

4.1.4. `mov`

```
int mov(int value, int x){
    return value; /* parmetro x    apenas placeholder */
}
```

Listing 5. Movimenta valor

Copia o valor para o registrador destino; o segundo argumento existe apenas para manter a assinatura uniforme.

4.1.5. `increment` / `decrement`

```
int increment(int reg, int x){
    return reg++;          /* retorna valor antes do ++ */
}
int decrement(int reg, int x){
    return reg--;          /* retorna valor antes do -- */
}
```

Listing 6. Incremento e decremento

Executam pós-incremento/pós-decremento. Se o comportamento desejado for o valor já atualizado, trocar por `++reg` / `--reg`.

4.1.6. `greater_than` / `less_than`

```
int greater_than(int regA, int regB){
    return (regA > regB) ? TRUE : FALSE;
}
int less_than(int regA, int regB){
    return (regA < regB) ? TRUE : FALSE;
}
```

Listing 7. Comparações

Retornam `TRUE` ou `FALSE` conforme a comparação aritmética.

4.1.7. `module`

```
int module(int regA, int regB){
    return regA % regB;
}
```

Listing 8. Módulo

Operação de resto da divisão inteira. **Observação:** não há verificação de divisor zero.

4.1.8. `and_function`

```
int and_function(int regA, int regB){
    return (regA == 1 && regB == 1) ? 1 : 0;
}
```

Listing 9. AND lógico simplificado

Implementa AND lógico específico para bits unitários.

4.2. Operadores do Algoritmo Genético

4.2.1. `double_to_bin`

```
void double_to_bin(Chromosome* chromosome){
    for (int i = 0; i < chromosome->size; i++)
        chromosome->bin_arr[i] = (chromosome->double_arr[i] >
            ↪ 0.5)
                                ? 1 : 0;
}
```

Listing 10. Quantização dos genes

Converte cada gene real (0–1) em bit usando limiar 0.5.

4.2.2. `mutation`

```
void mutation(Chromosome* c, int n_instr){
    int idx = rand() % (n_instr * 4);
    c->bin_arr[idx] ^= 1;     /* inverte o bit */
}
```

Listing 11. Mutação de 1 bit

Inverte um único bit aleatório por cromossomo.

4.2.3. `crossover`

```
Chromosome crossover(Chromosome* parent1, Chromosome* parent2
    ↪ , int n_instructions){
    Chromosome child;
    child.bin_arr = malloc(sizeof(int) * parent1->size);
    child.double_arr = malloc(sizeof(double) * parent1->size)
        ↪ ;
    child.fitness = 8;
    child.size = parent1->size;
    if(parent1->fitness > parent2->fitness){
        for(int i = 0; i < parent1->size; i++){
            child.bin_arr[i] = parent1->bin_arr[i];
            child.double_arr[i] = parent1->double_arr[i];
        }
    }else{
        for(int i = 0; i < parent2->size; i++){
            child.bin_arr[i] = parent2->bin_arr[i];
            child.double_arr[i] = parent2->double_arr[i];
        }
    }
    int j = n_instructions * 2;
    for(int i = 0; i < n_instructions * 2; i++){
        child.bin_arr[i] = parent1->bin_arr[i];
        child.bin_arr[j] = parent2->bin_arr[j];

        child.double_arr[i] = parent1->double_arr[i];
        child.double_arr[j] = parent2->double_arr[j];
        j++;
    }
    return child;
}
```

Listing 12. Cruzamento elitista

Copia integralmente o pai com melhor `fitness` e, em seguida, sobrescreve metade dos genes iniciais/finais para criar diversidade .

4.2.4. `selection`

```
Chromosome selection(Population* pop){
    int max_pop = pop->size;
    int num_instructions = pop->e->num_instructions;
    int index_chosen_parent = rand() % max_pop; //index
        ↪ chosen parent(from 0 to 9)
    int attempts = 15;
    while(pop->chromosomes[index_chosen_parent].fitness <= (2
        ↪  * num_instructions) && attempts < 50){
        index_chosen_parent = rand() % max_pop;
        attempts++;
```

```
    }

    Chromosome chrom = helper_selection(pop->chromosomes[
        ↪ index_chosen_parent], pop->chromosomes[
        ↪ index_chosen_parent].size);
    return chrom;
}
```

Listing 13. Seleção em torneio estocástico

Escolhe um pai cujo `fitness` seja razoável, evitando indivíduos muito fracos.

4.2.5. `fitness_func`

```
void fitness_func(Population* pop){
    Chromosome* chrom_pop = pop->chromosomes;
    int num_pop = pop->size;
    int num_instructions = pop->e->num_instructions;
    char* perfect_indiv = malloc(sizeof(char) * (
        ↪ num_instructions * 4 + 1));
    for(int i = 0; i < num_instructions; i++){
        if(i == 0)
        sprintf(perfect_indiv, "%s", pop->e->Instruc_arr[i].
            ↪ code);
        else
        sprintf(perfect_indiv + (4 * i), "%s", pop->e->
            ↪ Instruc_arr[i].code);
    }
    for(int i = 0; i < num_pop; i++){
        chrom_pop[i].fitness = 4 * num_instructions;
        for(int j = 0; j < num_instructions * 4; j++) {
            if((chrom_pop[i].bin_arr[j] + '0') !=
                ↪ perfect_indiv[j]){
                chrom_pop[i].fitness -= 1;
            }
        }
    }
    pop->best_fitness = pop->chromosomes[0].fitness;
    for(int i = 1; i < pop->size; i++){
        if(pop->best_fitness < pop->chromosomes[i].fitness){
            pop->best_fitness = pop->chromosomes[i].fitness;
        }
    }
    pop->best_chromosome = pop->chromosomes[0];
    for(int i = 1; i < pop->size; i++){
        if(pop->best_chromosome.fitness < pop->chromosomes[i
            ↪ ].fitness)
        pop->best_chromosome = pop->chromosomes[i];
    }
```

```
}
```

Listing 14. Avaliação de aptidão

A função fitness, através de distância de Hammig, calcula o quão longe cada indivíduo/cromossomo está de ser perfeito. Nos dois benchmarks testados, o fitness perfeito é igual a 8 devido a se tratar de 2 instruções de 4 bits.

4.2.6. `initialize_population`

```
void initialize_population(Population* pop, int size){
    for (int i = 0; i < pop->size; ++i){
        pop->chromosomes[i].size = size;
        /* aloca vetores e sorteia valores */
    }
}
```

Listing 15. População inicial aleatória

Preenche `double_arr` com números uniformes e deriva `bin_arr` coerente.

4.2.7. `genetic_alg`

```
void genetic_alg(Population* pop){
    printf("inside genetic algorithm\n");
    int flag = TRUE;
    double mutation_rate = (double)rand()/RAND_MAX;
    pop->generation = 1;
    while(pop->generation < 50 && flag == TRUE){
        printf("inside while\n");
        fitness_func(pop);
        print_pop_with_fitness(pop);
        if(pop->best_fitness == NUM_BITS * pop->e->
            ↪ num_instructions){
            printf("\n A perfect chromosome was found in the
                ↪ %d generation!!\n", pop->generation);
            print_chromosome(&pop->best_chromosome);
            flag = FALSE;
        }else{
            Chromosome parents[2];
            for(int i = 0; i < pop->size; i++){
                parents[0] = selection(pop);
                parents[1] = selection(pop);
                pop->chromosomes[i] = crossover(&parents[0],
                    ↪ &parents[1], pop->e->num_instructions);
                mutation_rate = (double)rand()/RAND_MAX;
                if(mutation_rate > 0.1 && mutation_rate <
                    ↪ 0.7){
```

```
                    mutation(&pop->chromosomes[i], pop->e->
                        ↪ num_instructions);
                }
            }
        }
        pop->generation++;
    }
}

}
```

Listing 16. Loop principal do GA

Coordena avaliação, seleção, cruzamento e mutação por até 50 gerações ou até encontrar a solução perfeita .

5. Benchmarks de Avaliação

Cada benchmark representa uma *tarefa-alvo* que o cromossomo deve aprender a executar. As funções `generate_F#` criam um objeto `Expression` cuja sequência de instruções codifica a solução "perfeita" para o problema. A aptidão de cada indivíduo é avaliada comparando seu programa com a string binária desses opcodes—vide `fitness_func` na Seção 4.

5.1. Benchmark F1 – Soma Simples

Definição. $D = A + B$

```
Expression* generate_f1(){
    //sum, mov
    printf("\nInside generate_f1\n");
    const char** ptr_f1 = f1;
    Expression* exp = malloc(sizeof(Expression));
    isMemoryAllocated(exp);
    exp->num_instructions = count_instructionsf1();
    exp->registers = malloc(sizeof(int) * NUM_REG);
    isMemoryAllocated(exp->registers);
    populate_instruc_arr(exp, ptr_f1);
    //sum
    exp->Instruc_arr[0].input_regs = malloc(sizeof(int*) *
        ↪ NUM_INPUTS);
    exp->Instruc_arr[0].input_regs[0] = &exp->registers[0];
        ↪ //registers[0] = regA
    exp->Instruc_arr[0].input_regs[1] = &exp->registers[1];
        ↪ //registers[1] = regB
    exp->Instruc_arr[0].output_reg = &exp->registers[0];

    //mov
    exp->Instruc_arr[1].input_regs = malloc(sizeof(int*) *
        ↪ NUM_INPUTS);
    exp->Instruc_arr[1].input_regs[0] = &exp->registers[0];
        ↪ //registers[0] = regA
```

```
20      exp->Instruc_arr[1].input_regs[1] = &exp->registers[1];
            ↪ //registers[1] = regB
21      exp->Instruc_arr[1].output_reg = &exp->registers[3];
22      print_registers_address(exp->Instruc_arr, exp->
            ↪ num_instructions);
23      return exp;
24  }
```

Listing 17. Função `generate_f1`

Explicação.

1. Aloca estrutura `Expression` e vetor de registradores (`R0–R3`).
2. Popula `Instruc_arr` com dois opcodes: `0000` (`add`) e `0011` (`mov`).
3. Configura ponteiros dos registradores de entrada/saída: `R0=A`, `R1=B`, resultado em `R3=D`.
4. O GA é considerado *perfeito* quando gera exatamente a string binária `00000011`.

5.2. Benchmark F2 – Soma Condicional

Definição. $D = (A + B > C)\ ?\ 1 : 0$

Operações requeridas: `add`, `greater_than`, `mov`. O gerador segue o mesmo padrão de `generate_f1`, trocando o bloco de duas instruções por três:

`add R0,R1` → `R0` `greater_than R0,R2` → `R0` `mov R0` → `R3`

Como resultado, o cromossomo alvo possui 3 opcodes (`0000 0110 0011`).

5.3. Benchmark F3 – Módulo

Definição. $D = A \bmod B$

Sequência mínima de duas instruções:

1. `module R0,R1 -> R0`
2. `mov  R0  -> R3`

Opcodes alvo: `1000 0011`.

5.4. Benchmark F4 – Incrementos Encadeados

Definição.

6. Execução do Programa

A Função `main()` orquestra todo o fluxo de execução — *do sorteio da população até a verificação final do cromossomo ótimo*. A Listagem 18 mostra o trecho relevante e o passo-a-passo é detalhado a seguir.

```
1   int main(){
2       srand((unsigned)time(NULL));                /* 1 */
3       Population pop;
4       pop.size = 50;                              /* 2 */
```

```
5       pop.chromosomes = malloc(sizeof(Chromosome)*pop.size);
6
7       /* === Sele  o do benchmark (hard-coded em 1) === */
8       int answer = 1;
9       switch (answer){
10          case 1:
11              chromosome_size = 16;              /* 3 */
12              initialize_population(&pop, chromosome_size);
13              pop.e = generate_f1();             /* 4 */
14              genetic_alg(&pop);                 /* 5 */
15
16              populate_registers(pop.e->registers, 2, 3, 4, 0);
17              /* 6: compara  o bit-a-bit + execu  o real */
18              ...
19              break;
20          /* demais cases reservados p/  F 2 F5  */
21      }
22      return 0;
23  }
```

Listing 18. Função principal simplificada

1. **Semente do RNG.** Garante reprodutibilidade das populações e operações GA (`rand()` é usado em `initialize_population`, `mutation`, `selection` e `crossover`).
2. **Alocação da população.** Cria um vetor de 50 cromossomos; cada um será preenchido em `initialize_population()` com genes aleatórios.
3. **Comprimento do cromossomo.** Para o benchmark F1 são 16 bits (2 instruções $\times$ 4 bits/opcode $\times$ 2 operandos).
4. **Geração da expressão-alvo.** `generate_f1()` devolve um ponteiro para `Expression` contendo a sequência perfeita `add + mov`. Nessa chamada, ponteiros dos registradores são antecipadamente ligados às entradas A, B e à saída D (vide Seção 5).
5. **Loop Genético.** `genetic_alg(&pop)` avalia a população; quando `best_fitness == 8` (máximo para 2 instruções) a busca encerra antes da 50ª geração.
6. **Validação funcional.**
   (a) Copia a string perfeita de opcodes para `chrom_benchmark`.
   (b) Compara bit a bit com o cromossomo vencedor (`best_chromosome`).
   (c) Executa cada instrução via ponteiro de função (`ptr_to_func`), atualizando o endereço apontado por `output_reg`.
   (d) Ao final, imprime o valor de `R3` (esperado: $D = A + B$ com $A = 2$ e $B = 3 \Rightarrow 5$).

Rotinas auxiliares.

- `print_population()` exibe genes em formato *double* e binário; útil para depuração visual.

- `isMemoryAllocated()` lança mensagem de erro caso a alocação via `malloc()` falhe.
- `populate_instruc_arr()` percorre `instruc_list[]` e replica metadados para a expressão, evitando *deep copies* de strings.
- `print_registers_address()` e `print_instruc_arr()` servem como "mapa" de ponteiros — garantem que cada entrada/saída esteja corretamente ligada.

Observação sobre MPI. O cabeçalho `<mpi.h>` já foi incluído para compatibilidade com as etapas futuras; entretanto, no momento não há chamadas MPI dentro de `main()`. A paralelização será introduzida na Etapa 3 usando a topologia Mestre–Escravo descrita na Seção **??**. Até lá, o código opera inteiramente em modo sequencial.

Referências