MC970/MO644 - Projeto

Paralelização do Algoritmo Genético para a resolução do problema Min-Sum K-Clustering com PThreads

Felipe Augusto de Castro e Silva, RA 145993
Rafael Grisotto e Souza, RA 192765

1 - O problema Min K-Clustering Sum

O problema Minimum K-Clustering Sum é NP-completo e considera um conjunto finito X de pontos no plano, uma distância $d(x,y) \in \Re$ para cada par $x, y \in X$, com todas as distâncias respeitando a regra de desigualdade dos triângulos. A partir disso, pretende-se encontrar uma partição de X em subconjuntos $C1, C2, ..., Ck$, de forma a minimizar as somas de todas as distâncias entre elementos em um mesmo subconjunto, portanto, sendo k o número de subconjuntos procurados, a meta é minimizar a seguinte expressão:

$$\sum_{i=1}^{k} \sum_{v1,\, v2 \in Ci} d(v1,\, v2)$$

2 - Algoritmo Genético

A literatura contém poucos exemplos relevantes de abordagem do problema Minimum K-Clustering Sum por algoritmos genéticos, com isso e levando em conta que algoritmos genéticos mostram bons resultados provados empiricamente numa série de problemas e que boas respostas demandam tempos de execução muito grandes foi decidido que tal abordagem paralelizada será explorada neste trabalho.

Alguns outros trabalhos oferecem uma explicação profunda sobre como funcionam os algoritmos genéticos, contudo é importante definir aqui qual será o algoritmo genético base utilizado neste trabalho.

Cada indivíduo da população é um vetor de inteiros em que cada posição corresponde a um ponto de entrada e que pode assumir valores entre 0 e K - 1, ou seja, o cluster que tal ponto está localizado. Para o algoritmo base, primeiro é definida uma população inicial aleatória de tamanho passado por parâmetro. Depois são executadas um número suficiente de iterações; para cada iteração, escolhemos aleatoriamente parte da população atual como pais da próxima população, combinamos os pais para a geração de novos indivíduos, mutamos os filhos de forma aleatória. Por último, são selecionados os melhores indivíduos para permanecerem na próxima população.

3 - Desafios

O maior desafio deste trabalho foi conseguir um speedup relevante para o algoritmo genético, devido sua organização: de maneira geral, podemos definir o algoritmo genético como um loop que é executado um grande número de vezes, em que cada vez, algumas etapas são performadas, de forma a gerar uma nova população, que lidere a uma resposta melhor que a já encontrada, mas que dê margem à diversidade, para que não a execução não fique parada em mínimos locais.

A primeira ideia foi a de paralelizar o loop principal do algoritmo, contudo, este é DO-ACROSS, pois a população atual depende da população anterior para ser gerada. Notamos que teríamos que buscar paralelizar cada etapa do algoritmo e que isso poderia não ser eficiente, pois como cada etapa é acessada muitíssimas vezes, o overhead para a criação e gestão de threads poderia superar seus possíveis benefícios.

A partir daqui, rodamos o profiling, que está descrito na seção 4 e observamos que a função que calcula o fitness de cada elemento da população era o grande gargalo da aplicação. Contudo, nossos problemas relacionados ao overhead se agravaram, pois algumas etapas do loop principal chamam um número considerável de vezes a função fitness e ficar criando nova threads poderia ser muito caro. Nossas previsões foram confirmadas quando tentamos paralelizar o código com OpenMP, na qual todos os nossos testes não lideravam speedups e o profiling nos mostrava que estávamos perdendo para o overhead das threads.

Tentamos uma abordagem com pthreads, que começou a se mostrar útil na utilização de instâncias consideradas grandes para o problema, aliado a um processador que suporta 8 threads. Depois disso, tentamos localizar outros trechos do código que poderiam ser paralelizados e foi identificado que a função de mutação continha um loop DO-ALL. Contudo, sua paralelização foi abandonada, pois nada contribuía com o speedup. Depois de alguns resultados razoáveis, decidimos seguir em frente com novas ideias de paralelização.

O que nos veio em mente é que apesar da maioria dos trechos que chamavam a função de fitness eram loops DO-ACROSS, a função que mais chamava o fitness (geração dos pais da nova população), continha um loop DO-ALL. Foram observados speedups para essa abordagem, mas ela se mostrou pior que somente o paralelismo da função fitness, pois ela demanda a utilização de locks.

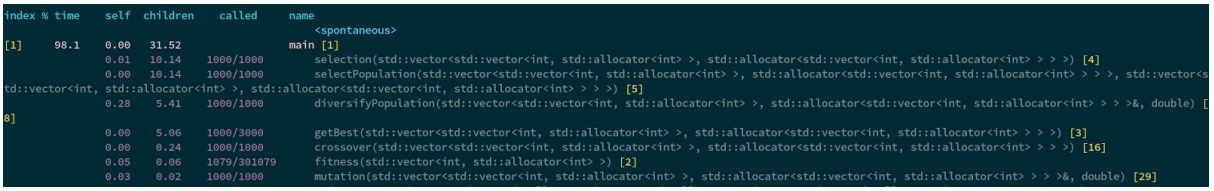
Por fim, um segundo problema foi abordado: é extremamente difícil encontrar bons parâmetros que liderem seu algoritmo genético à melhor resposta possível. Com isso, decidimos testar a criação de várias populações com vários parâmetros diferentes em um mesmo algoritmo, paralelizar suas execuções, fazendo com que a melhor resposta possa ser utilizada.

4 - Profiling

Utilizamos o pacote gprof e perf para observarmos as funções mais “quentes” do projeto. A partir do gprof, identificamos a função fitness pelo seu maior número de chamadas e a função selection, como são mostradas nos reports do gprof:

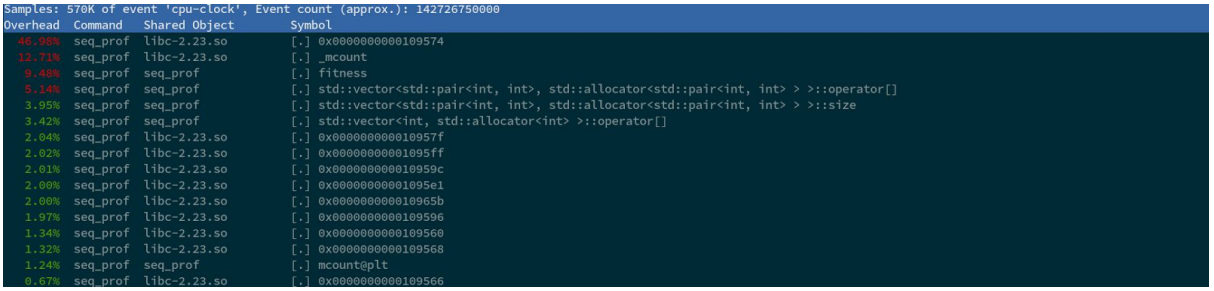
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 41.48     13.32    13.32   301079     0.04     0.10  fitness(std::vector<int, std::allocator<int> >)
 19.62     19.63     6.30 2916918392     0.00     0.00  std::vector<std::pair<int, int>, std::allocator<std::pair<int, int> > >::operator[](unsigned long)
 18.19     25.47     5.84 1587034370     0.00     0.00  std::vector<std::pair<int, int>, std::allocator<std::pair<int, int> > >::size() const
 14.33     30.07     4.60 3056331458     0.00     0.00  std::vector<int, std::allocator<int> >::operator[](unsigned long)
  1.87     30.67     0.60        1   600.20   600.20  std::vector<std::pair<int, int>, std::allocator<std::pair<int, int> > >::vector()
  1.81     31.25     0.58 364614799     0.00     0.00  __gnu_cxx::__enable_if<std::__is_integer<int>::__value, double>::__type std::sqrt<int>(int)
```



```
index % time    self  children    called     name
                                             <spontaneous>
[1]     98.1    0.00   31.52                 main [1]
                0.01   10.14    1000/1000        selection(std::vector<std::vector<int, std::allocator<int> >, std::allocator<std::vector<int, std::allocator<int> > > >) [4]
                0.00   10.14    1000/1000        selectPopulation(std::vector<std::vector<int, std::allocator<int> >, std::allocator<std::vector<int, std::allocator<int> > > >, std::vector<s
td::vector<int, std::allocator<int> >, std::allocator<std::vector<int, std::allocator<int> > > >) [5]
                0.28    5.41    1000/1000        diversifyPopulation(std::vector<std::vector<int, std::allocator<int> >, std::allocator<std::vector<int, std::allocator<int> > > >&, double) [
8]
                0.00    5.06    1000/3000        getBest(std::vector<std::vector<int, std::allocator<int> >, std::allocator<std::vector<int, std::allocator<int> > > >) [3]
                0.00    0.24    1000/1000        crossover(std::vector<std::vector<int, std::allocator<int> >, std::allocator<std::vector<int, std::allocator<int> > > >) [16]
                0.05    0.06    1079/301079      fitness(std::vector<int, std::allocator<int> >) [2]
                0.03    0.02    1000/1000        mutation(std::vector<std::vector<int, std::allocator<int> >, std::allocator<std::vector<int, std::allocator<int> > > >&, double) [29]
```

Na report do perf também é mostrado a função *fitness* como “quente”:



```
Samples: 570K of event 'cpu-clock', Event count (approx.): 142726750000
Overhead  Command   Shared Object        Symbol
  46.98%  seq_prof  libc-2.23.so         [.] 0x0000000000109574
  12.71%  seq_prof  libc-2.23.so         [.] _mcount
   9.48%  seq_prof  seq_prof             [.] fitness
   5.14%  seq_prof  seq_prof             [.] std::vector<std::pair<int, int>, std::allocator<std::pair<int, int> > >::operator[]
   3.95%  seq_prof  seq_prof             [.] std::vector<std::pair<int, int>, std::allocator<std::pair<int, int> > >::size
   3.42%  seq_prof  seq_prof             [.] std::vector<int, std::allocator<int> >::operator[]
   2.04%  seq_prof  libc-2.23.so         [.] 0x000000000010957f
   2.02%  seq_prof  libc-2.23.so         [.] 0x00000000001095ff
   2.01%  seq_prof  libc-2.23.so         [.] 0x000000000010959c
   2.00%  seq_prof  libc-2.23.so         [.] 0x00000000001095e1
   2.00%  seq_prof  libc-2.23.so         [.] 0x000000000010965b
   1.97%  seq_prof  libc-2.23.so         [.] 0x0000000000109596
   1.34%  seq_prof  libc-2.23.so         [.] 0x0000000000109560
   1.32%  seq_prof  libc-2.23.so         [.] 0x0000000000109568
   1.24%  seq_prof  seq_prof             [.] mcount@plt
   0.67%  seq_prof  libc-2.23.so         [.] 0x0000000000109566
```

Em suma, a partir dos dados do profiling identificamos as funções fitness e selection, porém pela característica de algoritmo genético, resolvemos paralelizar as populações também e como foi feito está na próxima seção.

5 - Como o código foi paralelizado

Função de fitness - a função de fitness é composta por dois loops encadeados (complexidade da função O(n^2)) e calcula a expressão que deve ser minimizada. Para sua paralelização, foi criada uma nova função chamada fit, do tipo ponteiro de void e que tinha como parâmetro também um ponteiro de void. Na função de fitness, são atribuídos um número uniforme de intervalos do loop mais externo que é executado por cada thread. A partir daí, uma struct com todos os parâmetros necessários é instanciada e passada para a função fit, que é atribuída e executada por uma thread, através da função pthread_create(). Por fim, esperamos que todas as threads terminem suas execuções através da função pthread_join(), somamos os cálculos de todas as threads em uma variável e retornamos para a função que a chamou.

```cpp
vector<int> cromossomeFit;
vector<vector<int> > popMut;

void *fit(void *aux){
    Aux *la = (Aux *) aux;
    int ans = 0;
    int l = la->l;
    int r = la->r;

    for(int i = l; i < r && i < points.size(); i++){
        for(int j = i + 1; j < points.size(); j++){
            if(cromossomeFit[i] == cromossomeFit[j]){
                ans += floor(0.5 + sqrt((points[i].first - points[j].first) * (points[i].first - points[j].first) + (points[i].second - points[j].second) * (points[i].second - points[j].second)));
            }
        }
    }

    return (void *) ans;
}

pthread_t threads[NUM_THREADS];
bool pala = false;

int fitness(vector<int> &cromossome){
    cromossomeFit = cromossome;

    for(int i = 0; i < NUM_THREADS; i++){
        Aux *aux = (Aux *) malloc(sizeof(aux));
        aux->l = i * points.size() / NUM_THREADS;
        aux->r = (i + 1) * points.size() / NUM_THREADS;

        if(i == NUM_THREADS - 1)
            aux->r = points.size();

        pthread_create(&threads[i], NULL, fit, (void *) aux);
    }

    int ans = 0;
    int st;

    for(int i = 0; i < NUM_THREADS; i++){
        pthread_join(threads[i], (void **) &st);
        ans += (int) st;
    }

    return ans;
}
```

Função de escolha de pais para a próxima população - nesta função, temos um loop que roda por um número de iterações dada como parâmetro que sorteia dois elementos da população atual (note que aqui utilizamos a função rand_r para gerar números aleatórios, já que a função rand() não é thread safe), escolhe o melhor através do cálculo de seus respectivos fitness e adiciona o melhor numa lista de pais. Para a paralelização desta função, criamos uma nova função chamada sel, do tipo ponteiro de void, que recebe um parâmetro também do tipo ponteiro de void. O número de iterações é dividida uniformemente pelo número de threads e a partir daí, uma struct com todos os parâmetros necessários é instanciada e passada para a função sel, que é atribuída e executada por uma thread, através da função pthread_create(). Por fim, esperamos que todas as threads terminem suas execuções através da função pthread_join() e retornamos a lista de pais. É importante ressaltar aqui que a lista de pais é uma variável global e só pode ter um acesso por vez, por isso, utilizamos as funções pthread_mutex_lock() e pthread_mutex_unlock() para organizar o acesso à lista e evitar qualquer erro.

```cpp
typedef struct{
    int esq, dir;
} Aux;

pthread_mutex_t locked;
unsigned int seed;

void *sel(void *aux){
    Aux *la = (Aux *) aux;
    int esq = la->esq;
    int dir = la->dir;

    for(int i = esq; i < dir; i++){
        int l = rand_r(&seed) % popAux.size();
        int r = rand_r(&seed) % popAux.size();

        if(fitness(popAux[l]) < fitness(popAux[r])){
            pthread_mutex_lock(&locked);
            parents.push_back(popAux[l]);
            pthread_mutex_unlock(&locked);
        } else {
            pthread_mutex_lock(&locked);
            parents.push_back(popAux[r]);
            pthread_mutex_unlock(&locked);
        }
    }
}

vector<vector<int> > selection(vector<vector<int> > pop){
    parents.clear();
    popAux = pop;
    pthread_t threads[NUM_THREADS];

    for(int i = 0; i < NUM_THREADS; i++){
        Aux *aux = (Aux *) malloc(sizeof(aux));
        aux->esq = i * pop.size() / NUM_THREADS;
        aux->dir = (i + 1) * pop.size() / NUM_THREADS;

        pthread_create(&threads[i], NULL, sel, (void *) aux);
    }

    for(int i = 0; i < NUM_THREADS; i++){
        pthread_join(threads[i], NULL);
    }

    return parents;
}
```

Várias populações - como nos casos anteriores, é criada uma nova função e aqui adicionamos o loop principal do algoritmo genético a ela. Cada set de parâmetros passado como entrada é atribuído a uma thread. A partir daqui, uma struct com todos os parâmetros necessários é instanciada e passada para a função exe, que é atribuída e executada por uma thread, através da função pthread_create(). Por fim, esperamos que todas as threads terminem suas execuções através da função pthread_join() e printamos a melhor resposta de todas as populações.

```
typedef struct{
    int P, generations, sizePopulation;
    double mutationRate, divRate;
} AuxExe;

void *exe(void *aux){
    AuxExe *la = (AuxExe *) aux;
    int generations = la->generations;
    int sizePopulation = la->sizePopulation;
    int P = la->P;
    double mutationRate = la->mutationRate;
    int divRate = la->divRate;
    int bestSol = INT_MAX;
    vector<vector<int> > pop = initialPopulation(sizePopulation, P);

    for(int i = 0; i < generations; i++){
        vector<vector<int> > parents = selection(pop);

        vector<vector<int> > cross = crossover(parents);

        mutation(cross, mutationRate);

        pop = selectPopulation(pop, cross);

        diversifyPopulation(pop, divRate);

        int idx = getBest(pop);

        if(bestSol > fitness(pop[idx])){
            bestSol = fitness(pop[idx]);
        }
    }

    return (void *) bestSol;
}
```

```
    pthread_t threads[numPopulations];

    for(int i = 0; i < numPopulations; i++){
        scanf("%d %lf %lf %d", &generations, &mutationRate, &divRate, &multSize);
        generations = 10;

        sizePopulation = ceil(log2(points.size())) * multSize;

        if(sizePopulation % 2 == 1) sizePopulation++;

        mutationRate = mutationRate / (double) points.size();

        AuxExe *aux = (AuxExe *) malloc(sizeof(AuxExe));
        aux->P = P;
        aux->generations = generations;
        aux->mutationRate = mutationRate;
        aux->divRate = divRate;
        aux->sizePopulation = sizePopulation;

        pthread_create(&threads[i], NULL, exe, (void *) aux);
    }

    int ans = INT_MAX, st;

    for(int i = 0; i < numPopulations; i++){
        pthread_join(threads[i], (void **) &st);

        // cout << (int) st << endl;

    }
```

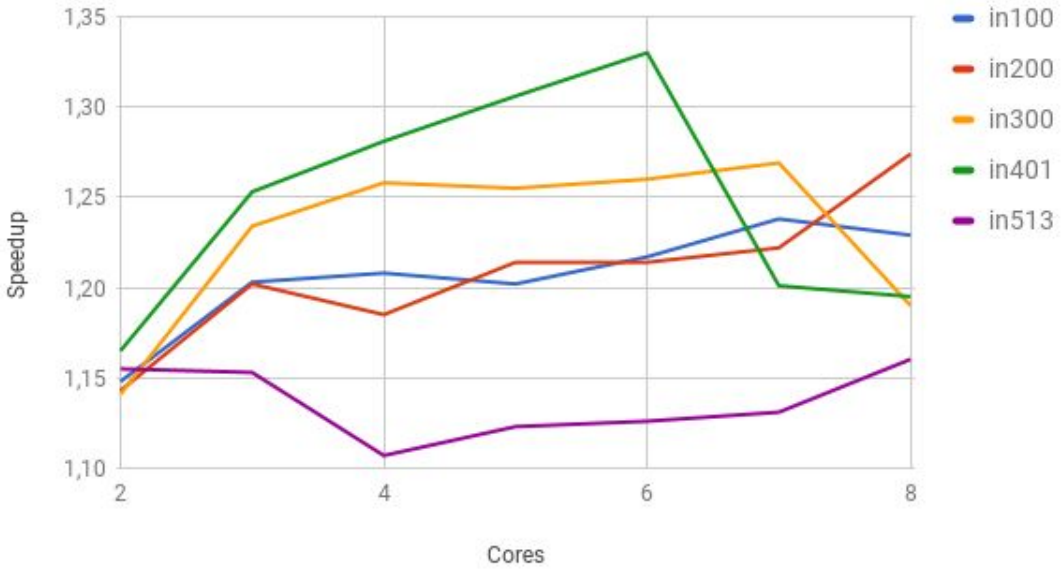
6 - Gráfico de speedup

Fizemos experimentos para os três tipos de paralelização, segue abaixo as figuras:

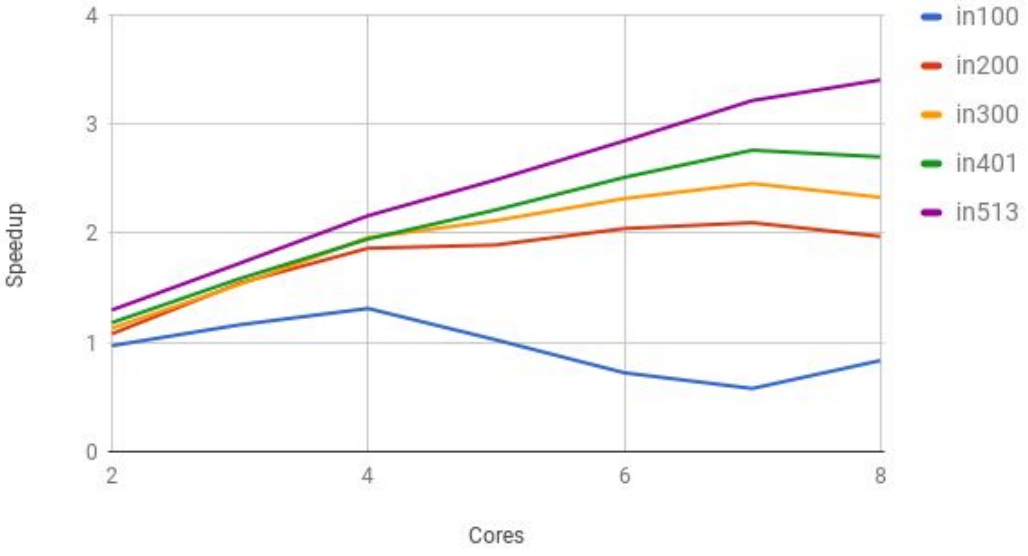
Para todas as formas de paralelização usamos 10 mil a quantidade de gerações para ficar mais real com seu uso prático na literatura, também definimos como 8 cores para o método de *Selection* e *Fitness*. Porém, para o método de múltiplas populações usamos 6 cores para executar 6 populações paralelamente e o serial 6 vezes com configurações dos parâmetros distintos.

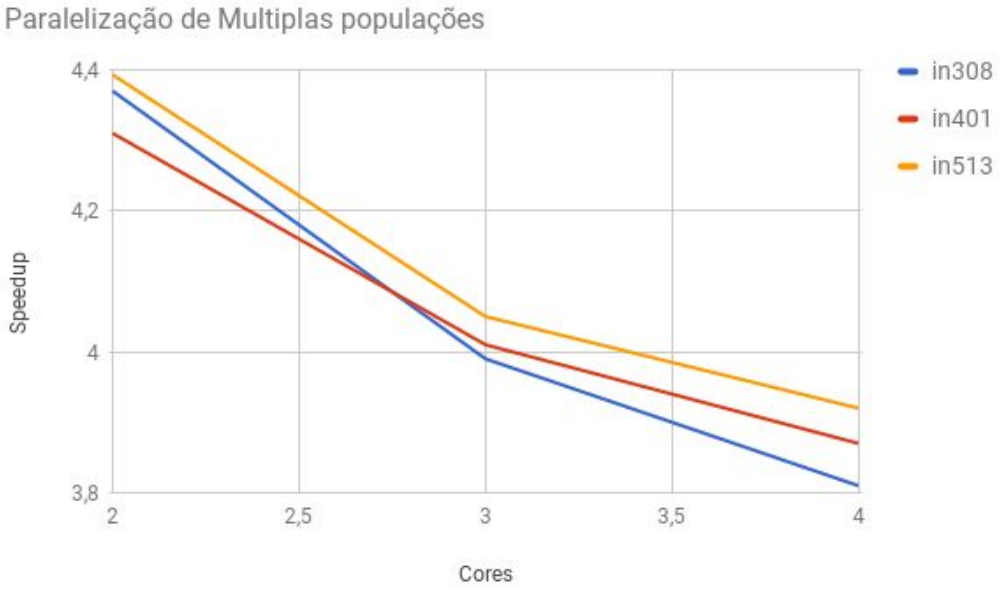
Podemos observar nas figuras que todas as instâncias tiveram formas de paralelização tiveram *speedup.* Destacando para o método de múltiplas populações e para a função *fitness* que tiveram *speedup* acima de 3.

Paralelização função Selection



Paralelização função Fitness




Paralelização de Multiplas populações
4,4
4,2
4
3,8
Speedup
2
2,5
3
3,5
4
Cores
in308
in401
in513