TÓPICOS ESPECIAIS EM INTERNET DAS COISAS "C"

Simulated Annealing Acoplado - Paralelização em linguagem C++ utilizando OpenMP

Leandro Silva Ferreira Junior
Lucas Lucena

1. Introdução

O Simulated Annealing Acoplado (CSA, do inglês Coupled Simulated Annealing) é um método estocástico de otimização global capaz de reduzir a sensibilidade dos parâmetros de inicialização enquanto guia o processo de otimização a execuções quaseótimas (XAVIER-DE-SOUZA et al., 2010). Ele é baseado no Simulated Annealing (KIRKPATRICK
et al., 1983) e no Minimizadores Locais Acoplados (CLM, do inglês Coupled Local Minimizers (SUYKENS; VANDEWALLE; MOOR, 2001). Nos CLMs, múltiplos otimizadores de descida de gradiente acoplado (do inglês coupled gradient descent optimizers) são utilizados porque são mais efetivos do que os otimizadores de descida de gradiente com múltiplo início (do inglês multistart gradient descent optimizers).

O CSA consiste em um conjunto de otimizadores SA cujo comportamento individual é similar à execução do Simulated Annealing, ou seja, os passos dos algoritmo que envolvem a geração e avaliação de uma única solução são executados individualmente por cada otimizador. Durante a geração de soluções, a temperatura de geração T_k^{gen} é responsável pelo grau de similaridade T_k^{gen} entre a solução corrente e a nova geração na iteração k. Assim, quanto maior o valor de , maior será a diferença média entre a nova solução e a corrente. A temperatura de aceitação T_k^{ac} utilizada durante a avaliação das nova soluções é responsável pela chance de aceitação de tais soluções, isto é, elevados valores resultam em maiores chances de uma nova solução ser aceita. Existe somente uma única temperatura de geração e de aceitação, independentemente no número de otimizadores.

2. Desenvolvimento

2.1 Implementações

2.1.1 Implementação Serial (linguagem C++)

Para a implementação em serial, o algoritmo recebe três parâmetros, o número de otimizadores, a dimensão do problema e o código da função a ser utilizada para avaliar os custos das soluções.

O algoritmo inicia definindo o número *k* de interações, a temperatura inicial de geração e de aceitação, e a variância desejada. Além disso declara variáveis, vetores e matrizes que serão utilizados no código. Após isso o relógio é inicializado.

```
int main(int argc, char* argv[])
{
    int       k                 = 0;                                          //Iterador
    double    tmp_gen           = 100.0;                                      //Temperatura de geração inicial
    double    tmp_ac            = 100.0;                                      //Temperatura de aceitação inicial
    int       m_threads         = strtol(argv[1], NULL, 10);                  //Número total de otimizadores
    int       n_size            = strtol(argv[2], NULL, 10);                  //Dimensão inicial
    int       func_num          = strtol(argv[3], NULL, 10);                  //CSA_EvalCost -> Function Number
    double    variancia         = 0.00;                                       //Variância
    double    max_custo_corrente = 0.0;                                       //Custo mais alto entre os otimizadores
    double    gamma             = 0.0;                                        //Termo de acoplamento gamma
    double    somatorio_A       = 0.0;                                        //Guarda o valor do somatoria das probabilidades de aceitação
    struct    timeval start, stop;                                            //Relógio
    double    custo_sol_melhor  = 0.0;
    double    m_threads_d       = static_cast<double>(m_threads);             //Número de threads em double
    double    variancia_d       = 0.99 * ((m_threads_d - 1)/pow(m_threads_d,2.0));  //Variância desejada

    gettimeofday(&start, NULL); //Start the clock
```

```
//Gera as matrizes de solução, os vetores de custo e das probabilidades de avaliação
double sol_corrente[m_threads][n_size];     //Solução corrente
double sol_nova [m_threads][n_size];       //Solução nova
double custo_sol_corrente[m_threads];
double custo_sol_nova[m_threads];
double A[m_threads];
```

Posteriormente as soluções iniciais são geradas de forma aleatória, com números entre -1 e 1, e o custo energético dessas soluções é avaliado através da função auxiliar *CSA_EvalCost*, que recebe como parâmetros o vetor solução, a dimensão da solução e o código da função de avaliação. Tudo isso feito dentro de um laço *for*, que simula uma região paralela com *m_threads* representando o número de otimizadores.

```
//Gera soluções iniciais - Xi
for (int i = 0; i < m_threads; ++i)
{
    for (int j = 0; j < n_size; ++j)
    {
        srand(i*time(NULL));
        num_aleatorio      = ((double) rand() / ((double)(RAND_MAX))); //Gera um número entre 0 e 1
        sol_corrente[i][j] = (2.0 * num_aleatorio) - 1.0;
    }
}

//Avalia o custo da solução corrente - X
for (int i = 0; i < m_threads; ++i)
{
    custo_sol_corrente[i] = CSA_EvalCost(sol_corrente[i], n_size, func_num);
```

Avaliamos o melhor custo e o custo máximo entre os otimizadores. Logo após, calculamos o termo de acoplamento *gamma* e as probabilidades *A* de aceitação, utilizando as fórmulas a seguir:

```
//Avalia GAMMA
for (int i = 0; i < m_threads; ++i)
{
    gamma += exp((custo_sol_corrente[i] - max_custo_corrente)/tmp_ac);
}

//Calcula a probabilidade de aceitação
for (int i = 0; i < m_threads; ++i)
{
    double e = exp((custo_sol_corrente[i] - max_custo_corrente)/tmp_ac);
    A[i] = e/gamma;
}
```

Nesse momento o algoritmo inicia o seu loop principal, em que vai gerar novas soluções, avaliar seus custos, testar se são melhores que as correntes e escalonar as temperaturas de geração e aceitação. Primeiramente geramos as novas soluções a partir das soluções correntes e avaliamos o seu custo. Verificamos então se o custo da nova solução é menor que o da solução corrente. Caso seja, trocamos a solução corrente pela nova e avaliamos novamente o melhor custo e o custo máximo entre os otimizadores.

```
//Iniciar o loop principal - critério de parada
for (k = m_threads; k < 1000000;)
{
    //Gera as novas soluções - y
    for (int i = 0; i < m_threads; i++)
    {
        for (int j = 0; j < n_size; ++j)
        {
            num_aleatorio  = ((double) rand() / ((double)(RAND_MAX))); //Gera um número entre 0 e 1
            sol_nova[i][j] = fmod(sol_corrente[i][j] + (tmp_gen*(tan(PI * (num_aleatorio - 0.5)))), 1.0);
        }
    }

    //Avalia o custo da solução nova - y
    for (int i = 0; i < m_threads; ++i)
    {
        custo_sol_nova[i] = CSA_EvalCost(sol_nova[i], n_size, func_num);
        k++;
    }
```

Caso a nova solução não seja melhor do que a corrente, jogamos a moeda, ou seja, geramos um número aleatório entre 0 e 1, e verificamos se a probabilidade de aceitação *A* é maior que esse número aleatório, se for realizamos a troca e a nova passa a ser a corrente, se não a corrente continua a mesma.

```
//Joga a moeda
for (int i = 0; i < m_threads; ++i)
{
    num_aleatorio  = ((double) rand() / ((double)(RAND_MAX)));

    if (custo_sol_corrente[i] > custo_sol_nova[i] or A[i] > num_aleatorio)
    {
        for (int j = 0; j < n_size; ++j)
        {
            sol_corrente[i][j] = sol_nova[i][j]; //Atualiza a solução corrente com o valor da solução nova
        }

        custo_sol_corrente[i] = custo_sol_nova[i]; //Atualiza o custo da solução corrente com o custo da solução nova

        //Verifica se a nova solucao corrente eh a melhor
        if (custo_sol_corrente[i] < custo_sol_melhor)
        {
            custo_sol_melhor = custo_sol_corrente[i];
        }

        //Avalia o melhor custo e o custo máximo entre os otimizadores
        if (i == 0)
        {
            max_custo_corrente = custo_sol_corrente[i];
        }
        else
        {
            if (custo_sol_corrente[i] > max_custo_corrente)
            {
                max_custo_corrente = custo_sol_corrente[i];
            }
        }
    }
}
```

Então reavaliamos o termo de acoplamento *gamma* e recalculamos a probabilidade de aceitação A, para então calcularmos o somatório de A ao quadrado. a partir desse somatório realizamos o cálculo da variância.

```
//Reavalia GAMMA
gamma = 0.00;
for (int i = 0; i < m_threads; ++i)
{
    gamma += exp((custo_sol_corrente[i] - max_custo_corrente)/tmp_ac);
}

//Recalcula a probabilidade de aceitação
for (int i = 0; i < m_threads; ++i)
{
    double e = exp((custo_sol_corrente[i] - max_custo_corrente)/tmp_ac);
    A[i] = e/gamma;
}

//Somatório de A
somatorio_A = 0.0;
for (int i = 0; i < m_threads; ++i)
{
    somatorio_A += pow(A[i], 2.0);
}

variancia = ((1/(m_threads_d))*(somatorio_A)) - (1/(pow(m_threads_d, 2.0)));
```

Se a variância for menor que a variância desejada, escalonamos a temperatura de aceitação para baixo, se for maior, para cima. Por fim, escalonamos a temperatura de geração, reiniciando o loop principal até o iterador *k* chegar a 1 milhão. O tempo gravado no txt e o melhor custo é impresso.

```
//Atualiza as temperaturas de aceitação e geração
if (variancia < variancia_d)
{
    tmp_ac = 0.95 * tmp_ac;
}
else
{
    if (variancia >= variancia_d)
    {
        tmp_ac = 1.05 * tmp_ac;
    }
}
tmp_gen = 0.99992 * tmp_gen;
```

2.1.2 Implementação Paralela (Utilizando OpenMP):

Para a implementação em paralelo, o algoritmo recebe os mesmos parâmetros que em serial. A grande diferença entre os dois algoritmos é que em paralelo utilizamos apenas vetores, não precisamos de uma matriz de soluções ou de laços simulando os otimizadores. Agora cada otimizador é um processador e temos divisão entre região paralela e região serial. Todas as variáveis são compartilhadas, com exceção do *k* que é privado, de *best* e *media* que são utilizadas apenas no final do código.

```
int main(int argc, char* argv[])
{
    int       k                 = 1;                                          //Iterador
    double    tmp_gen           = 100.0;                                      //Temperatura de geração inicial
    double    tmp_ac            = 100.0;                                      //Temperatura de aceitação inicial
    int       m_threads         = strtol(argv[1], NULL, 10);                  //Número total de otimizadores
    int       n_size            = strtol(argv[2], NULL, 10);                  //Dimensão inicial
    int       func_num          = strtol(argv[3], NULL, 10);                  //CSA_EvalCost -> Function Number
    double    variancia_d       = 0.99 * ((m_threads - 1)/pow(m_threads,2.0)); //Variância desejada
    double    variancia         = 0.00;                                       //Variância
    double    custos_correntes [m_threads];                                   //Vetor com os custos de cada otimizador
    double    max_custo_corrente  = 0.0;                                      //Custo mais alto entre os otimizadores
    double    gamma             = 0.0;                                        //Termo de acoplamento gamma
    double    best_sol [m_threads];                                           //Vetor que guarda os melhores custos_ de cada thread
    double    best              = 0.0;                                        //Melhor solução geral
    double    media             = 0.0;                                        //Média dos custos
    double    probabilidades_A [m_threads];                                   //Vetor que guarda o A de cada otimizador
    double    somatorio_A       = 0.0;                                        //Guarda o valor do somatoria das probabilidades de aceitação
    struct    timeval start, stop;                                            //Relógio
    int       modular           = 1000000 % m_threads;                        //Calcula se 1M é divisivel pelo numero de threads
    int       parada            = (1000000/(m_threads)) + modular;  //Calcula a condição de parada do laço principal
    double    m_threads_d       = static_cast<double>(m_threads);  //Número de threads em double


    gettimeofday(&start, NULL); //Start the clock
```

Abrimos a região paralela e cada otimizador gera suas soluções iniciais de forma aleatória entre -1 e 1, avalia o custo das soluções através da função auxiliar *CSA_EvalCost* e cada otimizador coloca o seu custo em um vetor de custos, em que cada otimizador só acessa a posição denotada pelo seu rank.

```
#pragma omp parallel num_threads(m_threads) default(none) shared(m_threads_d, tmp_gen, tmp_ac, m_th
{
    int    my_rank = omp_get_thread_num();                          //Ranking de cada otimizador
    double sol_corrente [n_size];                                   //Solução Corrente
    double sol_nova [n_size];                                       //Solução Nova
    double A = 0.0;                                                 //Probabilidade de aceitação
    double custo_sol_corrente, custo_sol_nova, custo_sol_melhor; //Custos das soluções

    //Semente
    struct drand48_data buffer;
    srand48_r(my_rank*time(NULL),(&buffer)); //Gera semente

    double num_aleatorio = 0.0;

    //Gera soluções iniciais - Xi
    for (int i = 0; i < n_size; i++)
    {
        drand48_r(&buffer, &num_aleatorio); //Gera um número entre 0 e 1
        sol_corrente[i] = (2.0 * num_aleatorio) - 1.0;
    }

    //Avalia o custo da solução corrente - X
    custo_sol_corrente = CSA_EvalCost(sol_corrente, n_size, func_num);
    custo_sol_melhor   = custo_sol_corrente;

    //Coloca o valor do custo da solução corrente no vetor com todos os custos de cada otimizador
    custos_correntes[my_rank] = custo_sol_corrente;
```

Então colocamos uma barreira, para garantir que o algoritmo só irá prosseguir após todos os otimizadores terem seus custos, e apenas um otimizador avalia o termo de acoplamento *gamma*.

```
#pragma omp barrier
//Avaliação de gamma
#pragma omp single
{
    //Calcula o custo máximo entre os otimizadores
    for (int i = 0; i < m_threads; ++i)
    {
        if (i == 0 )
        {
            max_custo_corrente = custos_correntes[i];
        }
        else
        {
            if (custos_correntes[i] > max_custo_corrente)
            {
                max_custo_corrente = custos_correntes[i];
            }
        }
    }

    //Avalia GAMMA
    for (int i = 0; i < m_threads; ++i)
    {
        gamma += exp((custos_correntes[i] - max_custo_corrente)/tmp_ac);
    }
}
```

Cada otimizador então calcula a sua probabilidade de aceitação *A* e colocamos outra barreira, garantindo que o algoritmo só prossegue após todos os otimizadores terem um *A* calculado.

```
//Calcula a probabilidade de aceitação
double e = exp((custo_sol_corrente - max_custo_corrente)/tmp_ac);
A = e/gamma;
probabilidades_A[my_rank] = A;
#pragma omp barrier
```

O loop principal é iniciado, com *k* partindo de *m_threads* até *parada*. Então cada otimizador gera novas soluções e avalia o custo delas com a função *CSA_EvalCost.* Como no algoritmo em serial, avaliamos se a nova é melhor que a corrente, caso seja trocamos a corrente pela nova, caso não avaliamos se *A* é maior que um número aleatório entre 0 e 1, sendo *A* maior, trocamos a corrente pela nova, não sendo seguimos para as próximas instruções sem alterar a solução corrente.

```
//Iniciar o loop principal - critério de parada
for (k = m_threads; k < parada;)
{
    //Gera as novas soluções - y
    for (int i = 0; i < n_size; i++)
    {
        drand48_r(&buffer, &num_aleatorio); //Gera um número entre 0 e 1
        sol_nova[i] = fmod(sol_corrente[i] + (tmp_gen*(tan(PI * (num_aleatorio - 0.5)))), 1.0);
    }

    //Avalia o custo da nova solução - y
    custo_sol_nova = CSA_EvalCost(sol_nova, n_size, func_num);
    k++;

    //Joga a moeda
    drand48_r(&buffer, &num_aleatorio); //Gera um número entre 0 e 1
    if (custo_sol_nova < custo_sol_corrente or A > num_aleatorio)
    {
        for (int i = 0; i < n_size; ++i)
        {
            sol_corrente[i] = sol_nova[i]; //Atualiza a solução corrente com o valor da solução nova
        }

        custo_sol_corrente        = custo_sol_nova;       //Atualiza o custo da solução corrente com o custo da solução nova
        custos_correntes[my_rank] = custo_sol_corrente;  //Atualiza o vetor com os custos correntes

        //Verifica se a nova solucao corrente eh a melhor
        if (custo_sol_corrente < custo_sol_melhor)
        {
            custo_sol_melhor = custo_sol_corrente;
        }
    }
```

Colocamos então mais uma barreira e entramos em outro single para reavaliar termo de acoplamento *gamma*. Então fechamos o single e cada otimizador recalcula o seu *A* individual. Colocamos outra barreira e abrimos outro single para calcular o somatório dos *A* ao quadrado e a variância.

```
//Atualização
#pragma omp barrier
#pragma omp single
{
    //Calcula o custo máximo entre os otimizadores
    for (int i = 0; i < m_threads; ++i)
    {
        if (i == 0 )
        {
            max_custo_corrente = custos_correntes[i];
        }
        else
        {
            if (custos_correntes[i] > max_custo_corrente)
            {
                max_custo_corrente = custos_correntes[i];
            }
        }
    }

    //Reseta o GAMMA
    gamma = 0.0;

    //Avalia GAMMA
    for (int i = 0; i < m_threads; ++i)
    {
        gamma += exp((custos_correntes[i] - max_custo_corrente)/tmp_ac);
    }
}
```

Calculamos a variância e, ainda dentro do single, atualizamos as temperaturas de aceitação e de geração. Chegamos então ao fim do laço, que irá se repetir até o *k* chegar em *parada.*

```
#pragma omp barrier

#pragma omp single
{
    //Somatorio das probabilidades de aceitação
    somatorio_A = 0;
    for (int i = 0; i < m_threads; ++i)
    {
        somatorio_A += pow(probabilidades_A[i], 2.0);
    }

    //Avalia a variância
    variancia = 0.0;
    variancia = ((1/(m_threads_d))*(somatorio_A)) - (1/(pow(m_threads_d, 2.0)));

    //Atualiza as temperaturas de aceitação e geração
    if (variancia < variancia_d)
    {
        tmp_ac = 0.95 * tmp_ac;
    }
    else
    {
        if (variancia >= variancia_d)
        {
            tmp_ac = 1.05 * tmp_ac;
        }
    }
    tmp_gen = 0.99992 * tmp_gen;
}
```

Então colocamos outra barreira e abrimos mais um single para parar de cronometrar o tempo de execução. O melhor custo de cada otimizador é então colocado no vetor *best_sol*. A região paralela é fechada, o melhor custo e a média dos custos são calculados e impressos, o tempo de execução é salvo no txt.

```
//Calcula a média e a melhor solução
for (int i = 0; i < m_threads; ++i)
{
    media = media + best_sol[i];

    if (i == 0)
    {
        best = best_sol[i];
    }
    else
        if (best_sol[i] < best)
        {
            best = best_sol[i];
        }
}
media = media/m_threads;
```

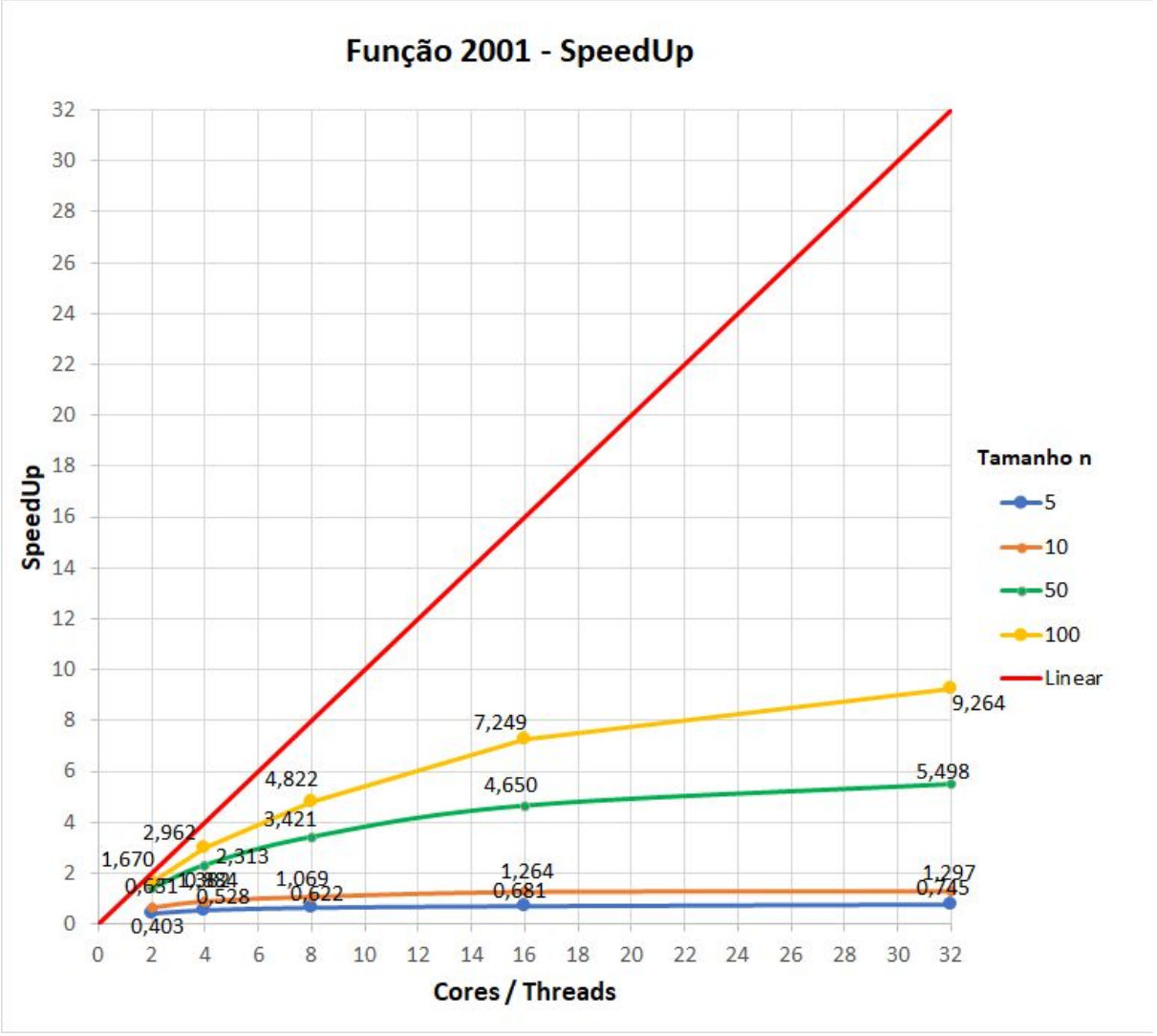
2.2. Resultados Experimentais

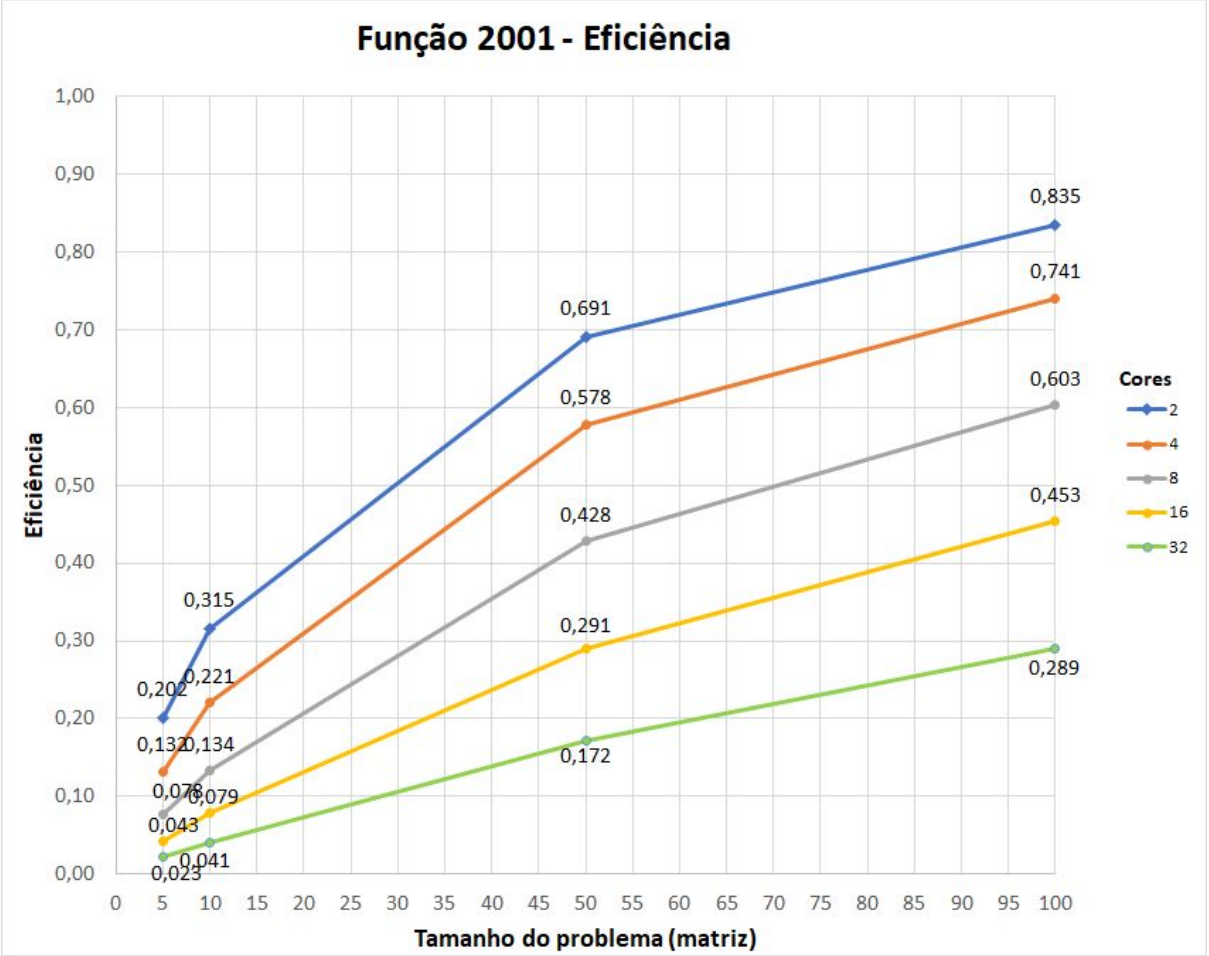
2.2.1 Configuração do problema

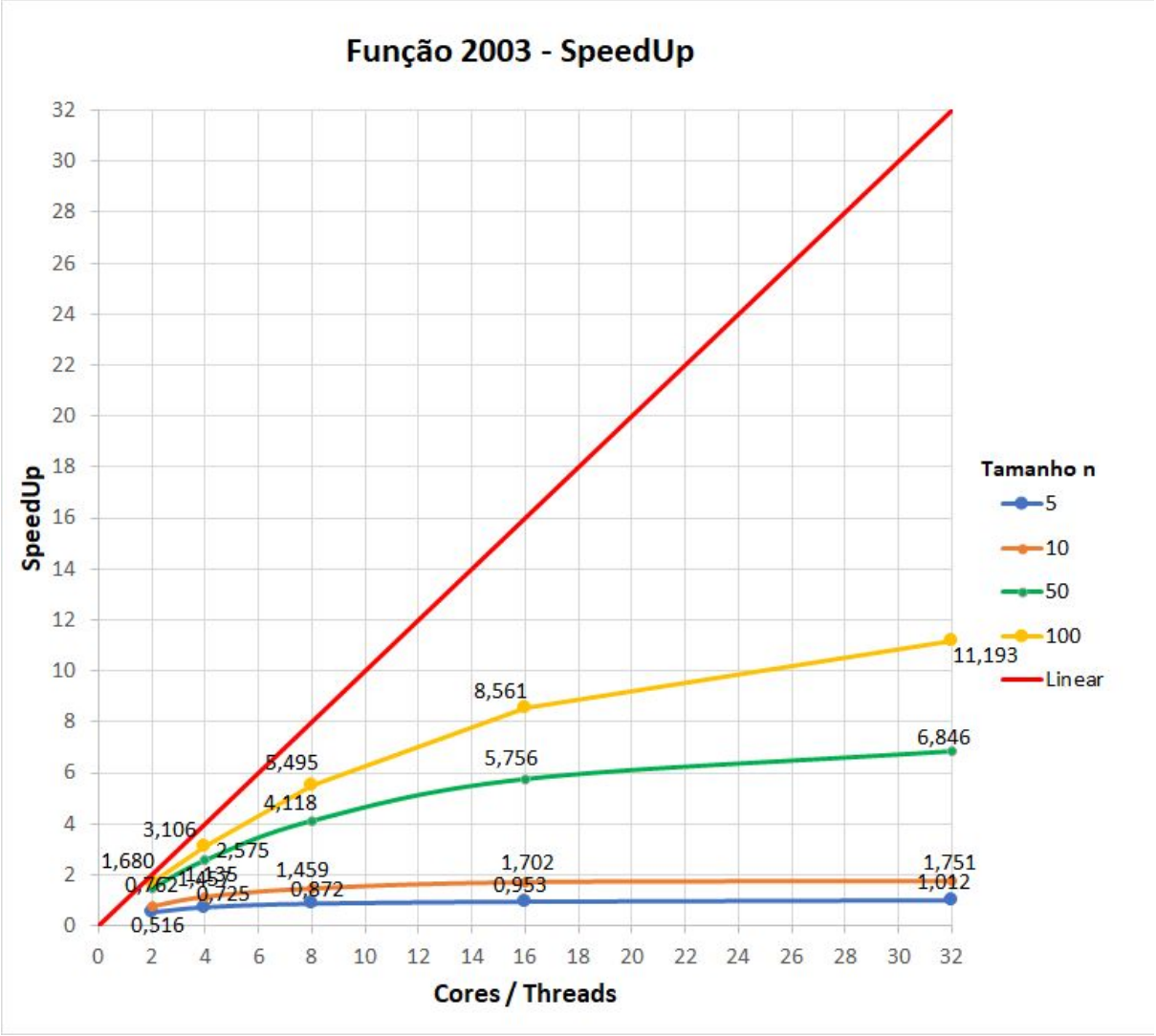
As dimensões das soluções foram definidas na descrição do trabalho. Deste modo, os tamanhos utilizados foram setados em 5, 10, 50 e 100. As funções foram definidas pelo .cpp auxiliar, *CSA_EvalCost,* temos três diferentes códigos para cada função, 2001, 2003 e 2006. Foi inserido um laço para o número de threads que serão utilizadas, desta forma, os testes para os 4 tamanhos serão realizados utilizando 2, 4, 8, 16 e 32 cores.

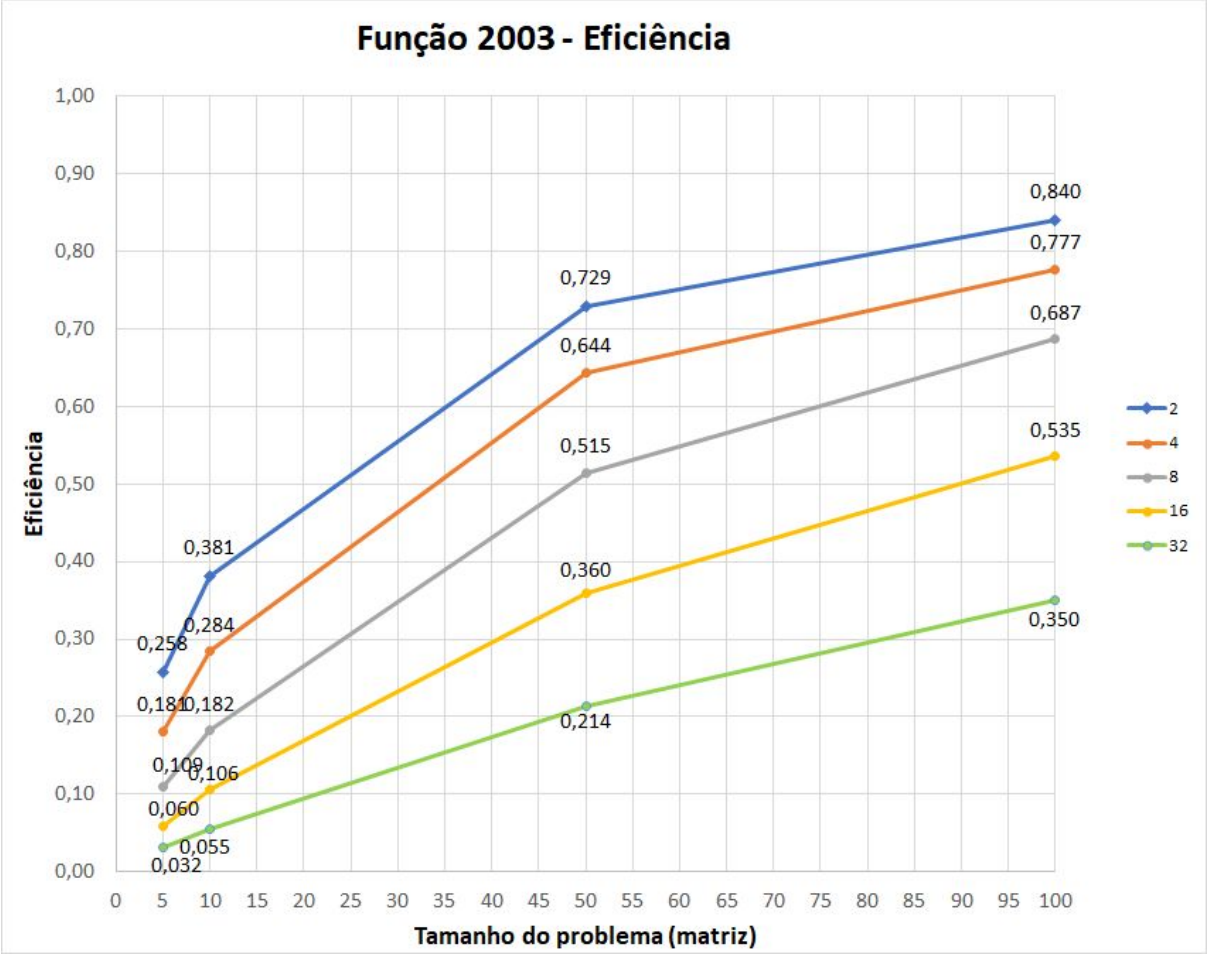
Os códigos tiveram suas execuções no supercomputador manipuladas por meio de um shellscript cuja configuração garante que a execução seja realizada 5 vezes para cada uma das 60 combinações entre dimensões e funções, gerando assim 300 tempos, os quais iremos utilizar para calcular o speedup e a eficiência do algoritmo. O shellscript também imprime no arquivo txt uma tabela com o número de cores utilizados, dimensão utilizada e código da função a cada 5 execuções, das quais o próprio programa escreve no mesmo arquivo o tempo de cada uma. No caso do script paralelo, realizamos 15 execuções, ao invés de 5.

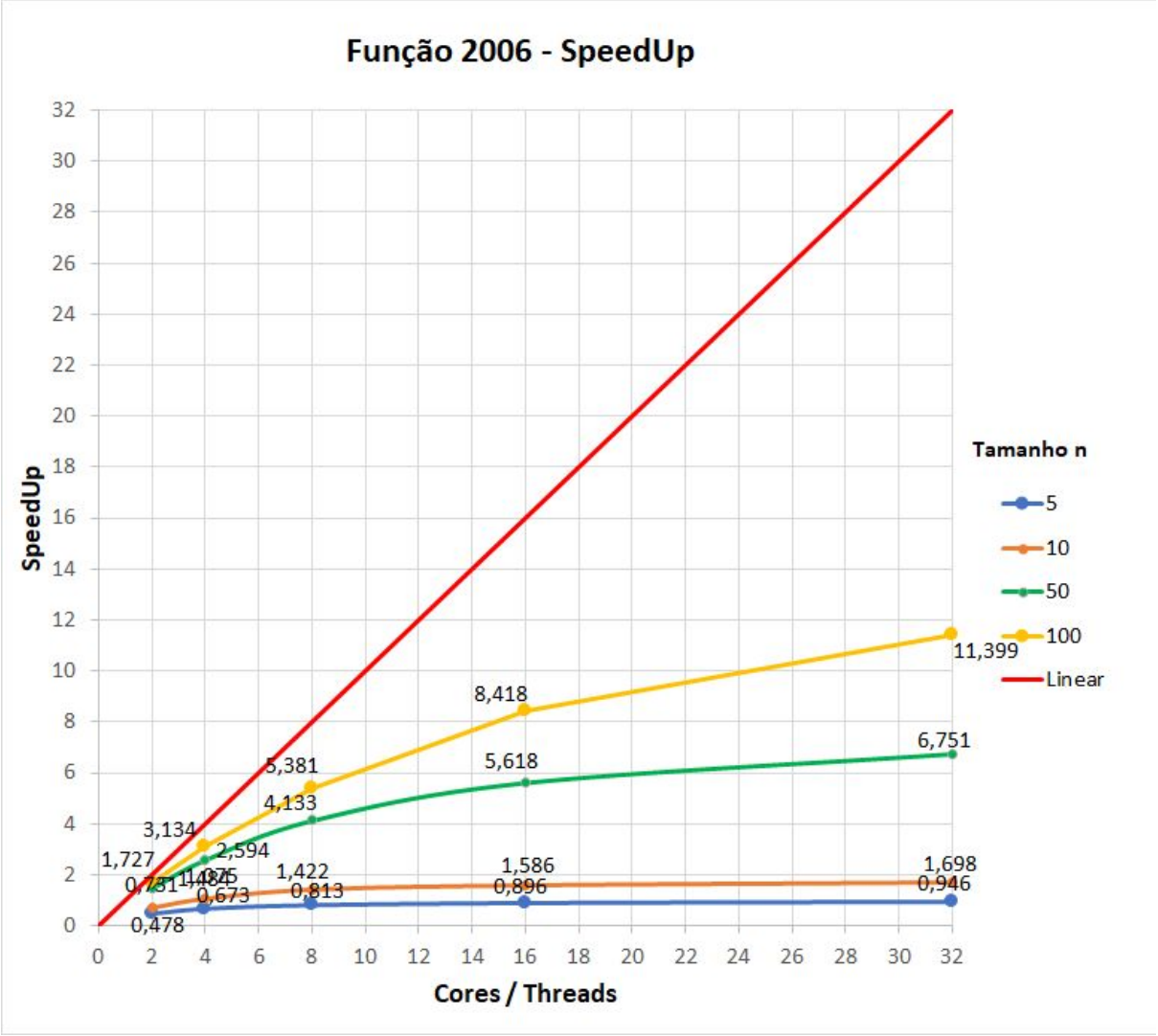
2.2.2 Análise de Escalabilidade (Speedup e Eficiência)

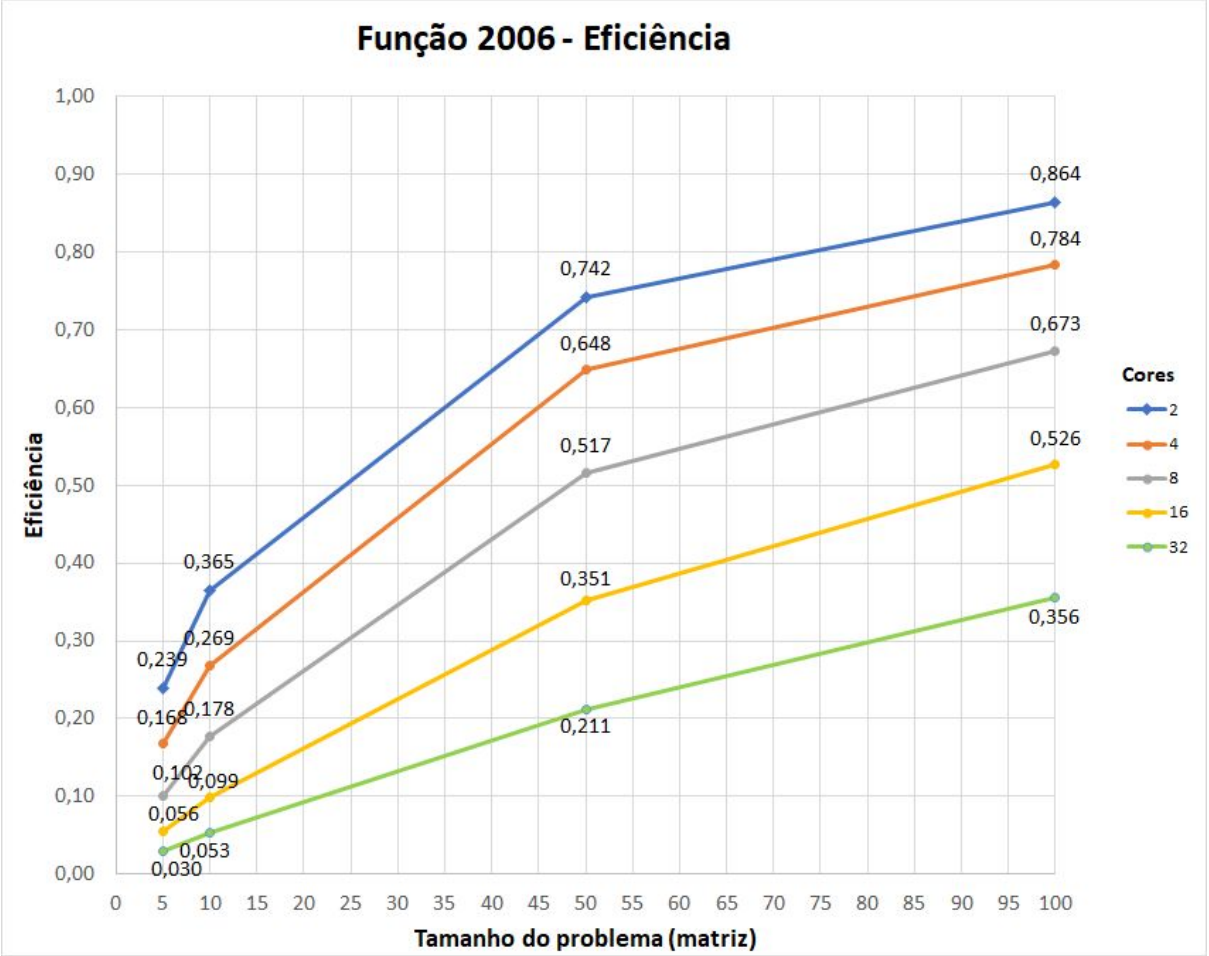



Função 2001 - Eficiência
Eficiência
Tamanho do problema (matriz)
Cores
2
4
8
16
32
0,835
0,741
0,603
0,453
0,289
0,691
0,578
0,428
0,291
0,172
0,315
0,221
0,134
0,079
0,041
0,202
0,132
0,078
0,043
0,023
1,00
0,90
0,80
0,70
0,60
0,50
0,40
0,30
0,20
0,10
0,00
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100


Função 2003 - SpeedUp
SpeedUp
Cores / Threads
Tamanho n
5
10
50
100
Linear
0,516
0,725
0,872
1,012
1,680
2,575
4,118
5,756
6,846
3,106
5,495
8,561
11,193
1,459
1,702
1,751
0,953


Função 2003 - Eficiência
Eficiência
Tamanho do problema (matriz)
1,00
0,90
0,80
0,70
0,60
0,50
0,40
0,30
0,20
0,10
0,00
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
0,840
0,777
0,729
0,687
0,644
0,535
0,515
0,381
0,360
0,350
0,284
0,258
0,214
0,182
0,181
0,109
0,106
0,060
0,055
0,032
2
4
8
16
32


Função 2006 - SpeedUp
SpeedUp
Cores / Threads
Tamanho n
5
10
50
100
Linear
0,478
0,673
0,813
0,896
0,946
1,727
2,594
4,133
5,618
6,751
3,134
5,381
8,418
11,399
1,422
1,586
1,698



4. Conclusão.

Podemos então concluir que o problema é *fracamente escalável*, pois conforme aumentamos o tamanho do problema, maior é a sua eficiência. Também percebemos que o speedup aumenta conforme aumentamos o tamanho do problema ou o número de cores , ou seja, quanto maior o tamanho do problema ou a quantidade de otimizadores, melhor será o desempenho do paralelo quando comparado ao serial.