Victor Gomes de Menezes

MULTIPLICAÇÃO DE MATRIZES QUADRADAS E COUPLED SIMULATED ANNEALING UTILIZANDO PROGRAMAÇÃO PARALELA COM OPENMP

Trabalho apresentado a matéria de Tópicos Especiais em Internet das Coisas do Instituto Metrópole Digital, na Universidade Federal do Rio Grande do Norte, com fim de obtenção de nota na 3° (segunda) unidade do semestre letivo de 2019.1.

Universidade Federal do Rio Grande do Norte – UFRN

Curso de Bacharelado em Tecnologia da Informação

Natal-RN

Junho de 2019

Sumário

1 Introdução

Este trabalho apresenta a implementação de dois programas, construídos na linguagem de programação C, que realizam a multiplição de duas matrizes quadradas. Também é desenvolvido uma outra aplicação, também feita em C, que implementa a metaheurística *Simulated Annealing Acoplado*(CSA). Todas as abordagens utilizam o *OpenMP*, uma interface de programação de aplicativo(API) para programação paralela explícita.

O problema da multiplicação de matrizes consiste em gerar aleatoriamente duas matrizes quadradas, realizar o produto entre elas e salvar os dados em uma matriz destinada ao resultado. Para isso, duas estratégias de resolução foram utilizadas: cada thread calcula todos os elementos de uma linha; cada thread calcula um único elemento aleatório por vez. Os dois programas construídos realizam, cada um, uma das duas tarefas.

O CSA é uma metaheurística baseada na têmpera de metais, tendo as temperaturas um papel muito importante no seu desenvolvimento. A implementação leva em consideração que cada thread aberta pelo *OpenMP* corresponde a um otimizador. Três funções foram escolhidas para sevirem de estudo. O intuito é encontrar o mínimo global de todas elas(0 para todas).

2 Desenvolvimento

2.1 Multiplicação de matrizes

2.1.1 Código Serial

A abordagem serial para este problema não envolve muitas etapas. Em resumo, o programa precisa gerar e alocar as matrizes de forma dinâmica para então fazer o cáculo de cada elemento individualmente.

O programa *serial_matrix_calc.c* foi inicializado com as variáveis necessárias. O trecho de código abaixo indica quais.

```
1     int i, j, k, result = 0;
2     int size_matrix;
3     int **a;
4     int **b;
5     int **c;
6     double total_time;
7     clock_t start, end;
```

2.1.1.1 Variáveis

As variáveis i, j e k são contadores usados nos loops para realizar o cálculo. A variável *result* armazena o valor atual do elemento $c_{i,j}$ calculado. O tamanho do problema, ou ainda, o número de linhas/colunas (a matriz é quadrada) das matrizes geradas foi armazenado na variável *size_matrix*. Os ponteiros de ponteiros a, b e c representam respectivamente as matrizes A, B e C utilizadas como exemplo na sessão **??**. A variável *total_time* representa o tempo total gasto pelo cálculo do programa, dado pela diferença entre as variáveis *end* e *start* do tipo *clock_t*.

2.1.1.2 Alocação dinâmica

O trecho de código abaixo mostra como foi feita a alocação dinâmica e o preenchimento das matrizes.

```
1     a = (int**)malloc(size_matrix * sizeof(int*));
2     b = (int**)malloc(size_matrix * sizeof(int*));
3     c = (int**)malloc(size_matrix * sizeof(int*));
4
5     for(i = 0; i < size_matrix; i++){
```

```
6           a[i] = (int*)malloc(size_matrix * sizeof(int));
7           b[i] = (int*)malloc(size_matrix * sizeof(int));
8           c[i] = (int*)malloc(size_matrix * sizeof(int));
9       }
10
11      srand(time(NULL));
12      for(i = 0; i < size_matrix; i++){
13          for (j = 0; j < size_matrix; j++) {
14              a[i][j] = rand() % 100;
15              b[i][j] = rand() % 100;
16              c[i][j] = -1;
17          }
18      }
```

2.1.1.3 Cáculo da matriz e tempo de execução

Por definição em sala de aula, o tempo de execução do programa seria apenas o necessário para realizar os cálculos dos elementos da matriz. Portanto, as variáveis que representam o início e o fim do tempo foram postas uma linha antes e uma linha depois do laço de execução dos cálculos, respectivamente. O tempo total é calculado na linha 15, realizando a diferença entre o fim e o início, além de dividir o valor pelo clock do sistema.

```
1       start = clock();
2
3       for (i = 0; i < size_matrix; i++) {
4           for (j = 0; j < size_matrix; j++) {
5               for (k = 0; k < size_matrix; k++) {
6                   result += a[i][k] * b[k][j];
7               }
8               c[i][j] = result;
9               result = 0;
10          }
11      }
12
13      end = clock();
14
15      total_time = ((double) (end - start)) / CLOCKS_PER_SEC;
```

O trecho de código entre as linhas 3 e 11 tem complexidade O(n^3). Para cada elemento calculado, é necessária a soma entre *size_matrix* elementos, por isso o terceiro laço é adicionado. O cálculo da multiplicação é explicado na sessão **??**.

Por fim, o tamanho do problema e seu tempo de execução são impressos em um arquivo de saída.

2.1.1.4 Execução do código

Um *Shell Script* chamado *script_serial_matrix_omp.sh* foi criado para compilar e executar o programa. São impressos alguns caracteres no arquivo gerado pelo programa. De antemão, o script remove e recria o arquivo para poder limpar o resultado de outras execuções anteriores.

```
1  #!/bin/bash
2  #SBATCH --partition=cluster
3  #SBATCH --job-name=matrix_multiplication_omp
4  #SBATCH --time=0-1:00
5  #SBATCH --output=serial_out_matrix_calc_omp.out
6  #SBATCH --error=serial_err_matrix_calc_omp.err
7  #SBATCH --nodes=1
8  #SBATCH --hint=compute_bound
9  #SBATCH --exclusive
10
11 rm ~/openmp/runtimes/serial_tempo_de_exec.txt
12 touch ~/openmp/runtimes/serial_tempo_de_exec.txt
13
14 gcc -g -Wall -o ~/openmp/serial_matrix_omp ~/openmp/src/serial_matrix_calc.
       c
15
16          tentativas=15
17
18          for size in 1500 1800 2000 2300
19          do
20                  echo -e "
                        ==================================================\n"
                        >> "/home/vgdmenezes/openmp/runtimes/
                        serial_tempo_de_exec.txt"
21                  for tentativa in $(seq $tentativas)
22                  do
23                          echo -e `/home/vgdmenezes/openmp/serial_matrix_omp
                                $size`
24                  done
25                  echo -e " " >> "/home/vgdmenezes/openmp/runtimes/
                        serial_tempo_de_exec.txt"
26                  echo -e "\n========================================" >>
                        "/home/vgdmenezes/openmp/runtimes/serial_tempo_de_exec.
                        txt"
27          done
28
29 exit
```

São executadas 15 tentativas para cada tamanho de problema definido, que no caso

são: 1500, 1800, 2000 e 2300. Os comandos *echo* são utilizados para fornecer uma melhor visualização da saída padrão do programa no arquivo *tempo_de_exec.txt*.

2.1.2 Código Paralelo

Nas duas abordagens escolhidas o preenchimento das matrizes deveria ser feito também de forma paralela. Para isso, um *parallel for* foi criado, dividindo as tarefas entre as threads. O *parallel for* foi usado nos dois programas. O trecho de código abaixo foi criado dentro da região paralela, logo após sua abertura.

```
1        # pragma omp parallel for default(none) firstprivate(size_matrix)
             private(i,j) shared(a,b,c)
2        for (i = 0; i < size_matrix; i++) {
3                for (j = 0; j < size_matrix; j++) {
4                a[i][j] = rand() % 100;
5                b[i][j] = rand() % 100;
6                c[i][j] = -1;
7            }
8        }
```

2.1.2.1 Cálculo por elementos

Nesta abordagem, cada thread criada irá pegar um elemento aleatório da matriz C e realizar sua multiplicação a partir das matrizes A e B. Quando todos os elementos forem calculados, as threads param e o programa tem seu fim.

As alocações dinâmicas e preenchimentos foram feitos exatamente iguais aos descritos na sessão 2.1.1.2.

O *OpenMP* fornece uma ferramenta que ajuda no processo de criação de tarefas que são executadas por todas as threads. O bloco de código **# pragma omp tasks** cria *tasks* para que sejam compartilhadas entre as threads do programa. Dentro do bloco, todo código escrito será replicado quantas vezes solicitado pelo programador.

No caso da multiplicação de matrizes por elementos aleatórios, cada elemento corresponde a uma *task*. O trecho de código abaixo mostra como foram criadas as *tasks*.

```
1        # pragma omp single
2        {
3            for (i = 0; i < size_matrix; i++) {
4                for (j = 0; j < size_matrix; j++) {
5                    /* One task for each element on C matrix */
6                    # pragma omp task firstprivate(i,j)
7                    {
```

```
8                   int k, mult_result = 0;
9                   for (k = 0; k < size_matrix; k++){
10                      mult_result += a[i][k] * b[k][j];
11                  }
12                  c[i][j] = mult_result;
13              } /* End task */
14          }
15      }
16    } /* End single */
17  }
18  total_time = omp_get_wtime() - start;
19
20  FILE *tempo;
21  tempo = fopen("/home/vgdmenezes/openmp/runtimes/tempo_de_exec_omp.txt",
        "a");
22  fprintf(tempo, "Problem Size = %d ----- Thread number = %d -----
      Runtime = %f\n", size_matrix, thread_count, total_time);
23  fclose(tempo);
```

O bloco single garante que somente uma thread entrará e executára os códigos seguintes. O primeiro laço for(na linha3) percorre as linhas da matriz e, o segundo(linha 4), as colunas, garantindo o percorrimento em todos os elementos. O bloco das *tasks* é aberto na linha 6 e dentro dele existe um outro laço que fará o cáculo do elemento $c_{i,j}$ expecífico. Na abertura do bloco, as variáveis i e j estão dentro da cláusula **firstprivate()**, guardando assim os seus valores antes da abertura.

Para cada elemento é feito em função das variáveis i e j, que, dentro de cada task, são fixas. A variável k representa o percorrimento de todos os elementos necessários para a multiplicação do valor $c_{i,j}$. O resultado da multiplicação é armazenado na variável *mult_result* e, após o laço, atribuído à sua devida posição na matriz C.

Por fim, o tempo total da execução é recuperado e impresso em um arquivo presente no supercomputador do IMD, onde foram realizados os experimentos.

2.1.2.2 Cálculo por linha

A abordagem de cálculo utilizando linhas ao invés de elementos aleatórios é muito semelhante a nível de código.

```
1     # pragma omp single
2     {
3       for (i = 0; i < size_matrix; i++) {
4           /* One task for each element on C matrix*/
5           # pragma omp task firstprivate(i)
6           {
```

```
7                       int j, k, mult_result = 0;
8                       for (j = 0; j < size_matrix; j++) {
9                           for (k = 0; k < size_matrix; k++){
10                              mult_result += a[i][k] * b[k][j];
11                          }
12                          c[i][j] = mult_result;
13                          mult_result = 0;
14                      }
15                  } /* End task */
16              }
17      } /* End single */
18  } /* End parallel zone */
19  total_time = omp_get_wtime() - start;
20
21  FILE *tempo;
22  tempo = fopen("/home/vgdmenezes/openmp/runtimes/tempo_de_exec_omp_lines.
        txt", "a");
23  fprintf(tempo, "Problem Size = %d ----- Thread number = %d ----- Runtime
        = %f\n", size_matrix, thread_count, total_time);
24  fclose(tempo);
```

A diferença se dá pelo posicionamento do laço sobre a variável k que entra no bloco da task, fazendo parte então da tarefa criada. Assim, ao invés da task representar 1 elemento, agora representa todos os elementos de 1 linha.

2.1.2.3 Execução do código

Por serem dois programas diferentes, decidi criar um script para cada(apenas um script daria conta da execução, mas por questões de organização escolhi esta abordagem). O script **script_matrix_omp.sh** executa o programa da seção 2.1.2.1, e o script **script_matrix_omp_lines.sh** o da seção 2.1.2.2. A mudança se dá apenas no executável e no arquivo escolhido para receber os outpus do programa. Abaixo está o script que executa um dos programas.

```
1  #!/bin/bash
2  #SBATCH --partition=cluster
3  #SBATCH --job-name=matrix_multiplication_omp
4  #SBATCH --output=out_matrix_calc_omp.out
5  #SBATCH --error=err_matrix_calc_omp.err
6  #SBATCH --nodes=1
7  #SBATCH --ntasks-per-node=32
8  #SBATCH --time=0-3:00
9  #SBATCH --exclusive
10 #SBATCH --hint=compute_bound
11
```

```
12 rm ~/openmp/runtimes/tempo_de_exec_omp.txt
13 touch ~/openmp/runtimes/tempo_de_exec_omp.txt
14
15 #Compila o codigo
16 gcc -g -Wall -fopenmp -o ~/openmp/matrix_calc_omp ~/openmp/src/
       matrix_calc_omp.c
17
18 tentativas=15
19
20 for size in 1500 1800 2000 2300
21 do
22         echo -e "
               ================================================================\n
               " >> "/home/vgdmenezes/openmp/runtimes/tempo_de_exec_omp.txt"
23         for thread in 2 4 8 16 32
24         do
25                 for tentativa in $(seq $tentativas)
26                 do
27                         echo -e `/home/vgdmenezes/openmp/matrix_calc_omp
                               $thread $size`
28                 done
29                 echo -e " " >> "/home/vgdmenezes/openmp/runtimes/
                       tempo_de_exec_omp.txt"
30         done
31         echo -e "\n
               ================================================================"
               >> "/home/vgdmenezes/openmp/runtimes/tempo_de_exec_omp.txt"
32 done
33
34 exit
```

Os tamanhos de problema 1500, 1800, 2000 e 2300 foram escolhidos para respeitar os limites de tempo impostos no trabalho, onde o menor tamanho deveria gerar em torno de 30 segundos de execução no código paralelo, e o maior, 90 segundos.

2.2 Coupled Simulated Annealing

Por definição serão usados multiplos otimizadores tanto na estratégia paralela, quando na serial. Os otimizadores têm um papel de aumentar as chances de chegar em um ótimo global. O objetivo é chegar em um valor mínimo, ou seja, o menor y(x) da função. Neste caso é sabido que os ótimos globais de todas as funções são iguais a 0, porém, geralmente esta informação não é conhecida.

2.2.1 CSA paralelo

De início duas funções são criadas para ajudarem na implementação. A função *maxValue* retorna o maior valor *double* em um array passado por parâmetro. A função *minValue* retorna o índice que corresponde ao menor valor dentro de um array passado por parâmetro. As duas funções são definidas abaixo.

```
1  double maxValue(double myArray[], int size) {
2      assert(myArray && size);
3      int i;
4      double maxValue = myArray[0];
5
6      for (i = 1; i < size; ++i) {
7          if ( myArray[i] > maxValue ) {
8              maxValue = myArray[i];
9          }
10     }
11     return maxValue;
12 }
13
14 double minValue(double myArray[], int size) {
15     assert(myArray && size);
16     int i, j = 0;
17     double minValue = myArray[0];
18
19     for (i = 1; i < size; i++) {
20         if ( myArray[i] < minValue ) {
21             minValue = myArray[i];
22             j = i;
23         }
24     }
25     return j;
26 }
```

2.2.1.1 Variáveis

As temperaturas de geração e aceitação, por definição, foram setadas em 100 no início do programa. O programa recebe por linha de comando o número de threads(que tratarei como número de otimizadores), a dimensão do problema e qual função será usada, respectivamente. Todas as variáveis são declaradas no início do programa, como descrito abaixo:

```
1  int main(int argc, char* argv[]) {
2
```

```
3       double t_gen = 100; // temperatura de geracao
4       double t_ac = 100; // temperatura de aceitacao
5       double num_aleatorio = 0.0; // numero aleatorio
6       double custo_sol_corrente, custo_sol_nova, melhor_custo; // energias
            corrente, nova e melhor
7       int dim; // dimensao do problema
8       int i; // iterador
9       double num_otimizadores = 0; // numero de threads/num_otimizadores
10      int num_function_choices[3] = {2001,2003,2006}; // opcoes de funcao
11      int num_function; // funcao selecionada
12      int avaliacoes = 0; // avaliacoes
13      int menor_custo_total; // indice do otimizador com melhor custo
14
15      /* atribuicoes iniciais */
16      num_otimizadores = (double) atoi(argv[1]); // pega a numero de threads
            do programa
17      dim = atoi(argv[2]); // pega a dimensao do arg da linha de comando
18      num_function = num_function_choices[atoi(argv[3])]; // pega a funcao a
            ser usada
19
20      double var_desejada = 0.99 * ((num_otimizadores - 1)/(num_otimizadores
            * num_otimizadores )); // calculo da variancia desejada
21      double *sol_corrente; // solucao corrente
22      double *sol_nova; // solucao nova
23      double *tmp = NULL; // usado para trocar valores
24      double *vetor_func_prob = (double *)malloc(num_otimizadores * sizeof(
            double)); // agammas
25      double *atuais_custos = (double *)malloc(num_otimizadores * sizeof(
            double)); // custos correntes de cada otimizador
26      double termo_acoplamento; // termo de acoplamento
27      double sigma;
28      double total_time; // tempo total de execucao
29      double start; // tempo de inicio da execucao
30
31      struct drand48_data buffer; // semente
```

2.2.1.2 Região paralela

Logo após a declaração das variáveis é aberta a região paralela, e logo de início são alocados os espaços para a solução corrente e solução nova de cada otimizador, além de ser iniciado o contador de tempo do programa. O rank da thread é atribuído a variável *my_rank*.

```
1       # pragma omp parallel num_threads((int)num_otimizadores) \
2       default(none) \
```

```
3     shared(start, total_time, vetor_func_prob, sigma, menor_custo_total, k,
          termo_acoplamento, avaliacoes, t_gen, t_ac, atuais_custos,
          num_otimizadores, dim, num_function, var_desejada) \
4     private(num_aleatorio, tmp, buffer, melhor_custo, custo_sol_corrente,
          sol_corrente, sol_nova, custo_sol_nova, i)
5     {
6         // apenas uma thread inicia o clock
7         # pragma omp single
8         {
9             start = omp_get_wtime();
10        }
11
12        sol_corrente = (double *)malloc(dim * sizeof(double)); // alocacao
              da solucao corrente
13        sol_nova = (double *)malloc(dim * sizeof(double)); // alocacao da
              nova solucao
14
15        int my_rank = omp_get_thread_num(); // rank da thread/otimizador
```

Então as primeiras soluções são geradas em cada otimizador a partir de números aleatórios entre -1.0 e 1.0, sendo sempre valores do tipo *double*.

Cada otimizador gera uma solução inicial, incrementa a avaliacao da funcao em 1, calcula sua energia, a atribui como solução corrente e insere a enegia corrente no array de atuais custos de todas as threads. O melhor custo do otimizador também recebe a energia corrente, pois ela é a primeira e única por enquanto. Após estes passos, a primeira barreira é feita para garantir que os otimizadores continuem apenas quando todos realizarem as tarefas.

```
1         srand48_r(time(NULL)+my_rank*my_rank,&buffer); // Gera semente
2         //Gera solu   es iniciais
3         for (i = 0; i < dim; i++){
4             drand48_r(&buffer, &num_aleatorio); //gera um n mero entre 0 e
                  1
5             sol_corrente[i] = 2.0*num_aleatorio-1.0;
6             // verifica se o valor esta no intervalo previsto
7             if (sol_corrente[i] < -1.0 || sol_corrente[i] > 1.0) {
8                 printf("Erro no limite da primeira solucao corrente: \n");
9                 exit(0);
10            }
11        }
12
13        custo_sol_corrente = CSA_EvalCost(sol_corrente, dim, num_function);
              // nova energia corrente
14
15        // incrementa +1 avaliacao na funcao objetivo
```

```
16          # pragma omp atomic
17              avaliacoes++;
18
19          // coloca a atual energia num vetor auxiliar de custos/energias
20          atuais_custos[my_rank] = custo_sol_corrente;
21
22          // melhor custo do otimizador, de inicio, eh o primeiro
23          melhor_custo = custo_sol_corrente;
24
25          // sincronizacao
26          # pragma omp barrier
```

O próximo passo é calcular o termo de acoplamento. Apenas uma thread fará o trabalho, já que o termo é uma variável global. Para isso, o bloco single foi criado, o que garante a exclusividade de uma thread.

```
1           // calculo do termo de acoplamento
2           # pragma omp single private(i)
3           {
4               termo_acoplamento = 0;
5               for (i = 0; i < (int) num_otimizadores; i++) {
6                   termo_acoplamento += pow(EULLER, ((atuais_custos[i] -
                        maxValue(atuais_custos, (int)num_otimizadores))/t_ac));
7               }
8           } // barreira implicita
9
10          // funcao de probabilidade de aceitacao
11          double func_prob = pow(EULLER, ((custo_sol_corrente - maxValue(
                atuais_custos, (int)num_otimizadores))/t_ac))/termo_acoplamento;
12          // verifica se a funcao esta no intervalo previsto
13          if (func_prob < 0 || func_prob > 1){
14              printf("Limite errado da funcao de probabilidade\n");
15              exit(0);
16          }
17          //adiciona o valor da funcao do otimizador N na sua posicao
                correspondente no vetor de funcoes
18          vetor_func_prob[my_rank] = func_prob;
```

O termo de acoplamento é calculado e logo após existe uma barreira implícita do single que garante a espera de todas as outras threads. Após o cálculo do termo, a função de probabilidade de aceitação(ou gamma) também é calculada por cada otimzador. Com o gamma calculado, seu valor é inserido na posição correspondente ao rank do otimizador(recuperado no início da região paralela).

Com o termo de acoplamento e o gamma calculados, o laço de repetição da região

paralela pode ser aberto. Nele, o primeiro passo é gerar uma nova solução a partir da criada anteriormente(e atribuída como solução corrente). A energia da nova solução é calculada e então a avaliação é incrementada em 1.

```
1     while(avaliacoes < 1000000){
2         // gera a nova solucao a partir da corrente
3         for (i = 0; i < dim; i++) {
4             drand48_r(&buffer, &num_aleatorio); //gera um n mero entre 0 e
                  1
5             sol_nova[i] = fmod((sol_corrente[i] + t_gen * tan(PI*(
                  num_aleatorio-0.5))), 1.0);
6             // verifica se o valor esta no intervalo previsto
7             if (sol_nova[i] > 1.0 || sol_nova[i] < -1.0) {
8                 printf("Intervalo de solu  es mal definido!\n");
9                 exit(0);
10            }
11        }
12
13        // nova energia
14        custo_sol_nova = CSA_EvalCost(sol_nova, dim, num_function);
15
16        // incrementa +1 avaliacao na funcao objetivo
17        # pragma omp atomic
18            avaliacoes++;
```

Como visto na linha 1, o critério de parada é de 1 milhão de avaliações à função objetivo, por isso é importante que cada otimizador incremente a variável toda vez que chamar a função.

O passo seguinte é avaliar a nova energia gerada a partir da nova solução.

```
1         // novo numero aleatorio entre 0 e 1
2         drand48_r(&buffer, &num_aleatorio);
3         if (num_aleatorio > 1 || num_aleatorio < 0) {
4             printf("ERRO NO LIMITE DE R = %f\n", num_aleatorio);
5             exit(0);
6         }
7
8         // avaliacao dos atuais custos/energias
9         if (custo_sol_nova <= custo_sol_corrente || func_prob >
              num_aleatorio){
10            tmp = sol_corrente;
11                sol_corrente = sol_nova;
12                sol_nova = tmp;
13            custo_sol_corrente = custo_sol_nova;
14            atuais_custos[my_rank] = custo_sol_nova;
```

```
15
16              // se a nova solucao for menor que a melhor/menor atual, troca
17              if (melhor_custo > custo_sol_nova) {
18                  melhor_custo = custo_sol_nova;
19              }
20          }
21
22          // sincronizacao
23          # pragma omp barrier
```

A avaliação consiste basicamente em checar se a nova energia é menor que a considerada energia corrente. Caso seja, a energia corrente se torna a nova. Caso não seja, então um outro critério é usado. Caso o gamma seja maior que um número aleatório entre 0 e 1, então também ocorre a troca de valores. O melhor custo também é verificado para garantir que o valor não se perca, caso seja melhor que o atual. Ao fim, mais uma barreira é colocada para esperar que todas as threads façam seus cálculos.

Com a energia corrente atualizada ou não, o termo de acoplamento deve ser novamente calculado. É usado o mesmo cálculo feito anteriormente. Esta parte é feita exclusivamente em um bloco single, isto pois o gamma será novamente calculado por cada thread logo após. Com esta configuração, o gamma se mantém atualizado com o novo valor do termo de acoplamento. Uma nova barreira é adicionada.

```
1           // calculo do termo de acoplamento
2           # pragma omp single private(i)
3           {
4               termo_acoplamento = 0;
5               for (i = 0; i < (int) num_otimizadores; i++) {
6                   termo_acoplamento += pow(EULLER, ((atuais_custos[i] -
                        maxValue(atuais_custos, (int) num_otimizadores))/t_ac));
7               }
8           } // barreira implicita
9
10          // recalcula a funcao de probabilidade
11          func_prob = pow(EULLER, ((custo_sol_corrente - maxValue(
                atuais_custos, (int) num_otimizadores))/t_ac))/termo_acoplamento
                ;
12          if (func_prob < 0 || func_prob > 1){
13              printf("Limite errado da funcao de probabilidade\n");
14              exit(0);
15          }
16          // adiciona novamente ao vetor de funcoes de probabilidade
17          vetor_func_prob[my_rank] = func_prob;
18
19          // sincronizacao
```

```
20          # pragma omp barrier
```

Um novo bloco single é aberto para a atualização das temperaturas de aceitação e geração.

```
1           # pragma omp single private(i)
2           {
3               // calculo da variancia da funcao de probabilidades de
                    aceitacao
4               sigma = 0;
5               for (i = 0; i < (int) num_otimizadores; i++) {
6                   sigma += (double) pow(vetor_func_prob[i], 2);
7               }
8               sigma = ((1/num_otimizadores) * sigma) - 1/(num_otimizadores *
                    num_otimizadores);
9
10              double sigma_limit = ((num_otimizadores - 1)/(num_otimizadores
                    * num_otimizadores));
11              if (sigma < 0 || sigma > sigma_limit){
12                  printf("Limite errado de sigma. sigma = %f, iteracao = %d\n
                        ", sigma, k);
13                  exit(0);
14              }
15
16              // avaliacao e atualizacao da temperatura de aceitacao
17              if (sigma < var_desejada){
18                  t_ac = t_ac * (1 - 0.01);
19              } else if (sigma > var_desejada){
20                  t_ac = t_ac * (1 + 0.01);
21              }
22
23              // atualizacao da temperatura de geracao
24              t_gen = 0.99992 * t_gen;
25
26          } // barreira implicita
27      }
```

O sigma é calculado a partir dos gammas de todas as threads e então são feitas as atualizações. Após o fim do bloco single, é encerrado o laço de repetição.

Caso o critério de parada seja alcançado, os melhores custos/energias dos otimizadores agora são tratados como atuais custos. Assim, o menor entre eles é escolhido e dado como output, junto com o tempo de execução.

3 Resultados e Análise

3.1 Multiplicação de Matrizes

O programa serial executou 15 tentativas para cada tamanho de problema. Os paralelos, 75 para cada tamanho de problema, sendo que esses 75 são divididos igualmente entre 2, 4, 8, 16 e 32 threads.

3.1.1 Speedups

3.1.1.1 Por linhas

O gráfico mostra um speedup interessante. Apenas o problema de tamanho 2000 gerou uma inconsistência ao rodar com 32 threads. Isto implicou negativamente no gráfico de eficiência do algoritmo.

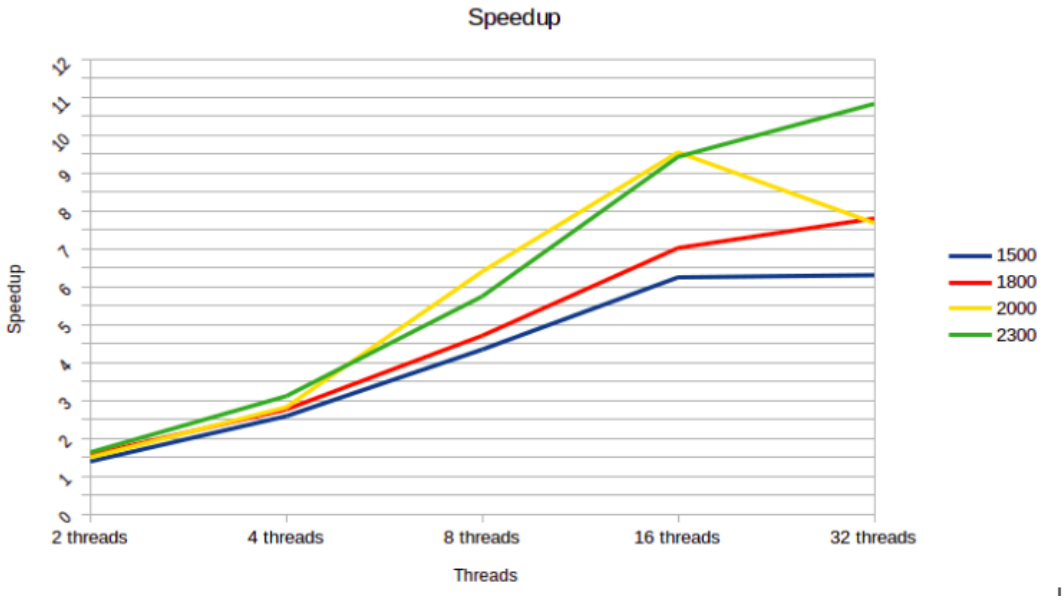

Figura 1 – Speedup da abordagem por linhas

3.1.1.2 Por Elementos

Os speedups deste programa tiveram uma quebra a partir de 16 threads, com exceção do problema 2300(mas que quebra em 32). Uma inconsistência na execução dos códigos no supecomputador gerou estes dados. Os runtimes estavam aumentando ao invés de diminuir, como se as threads não estivessem sendo usadas. Conversei com os representantes do NPAD em busca de soluções, mas não as encontrei a tempo de terminar o trabalho.

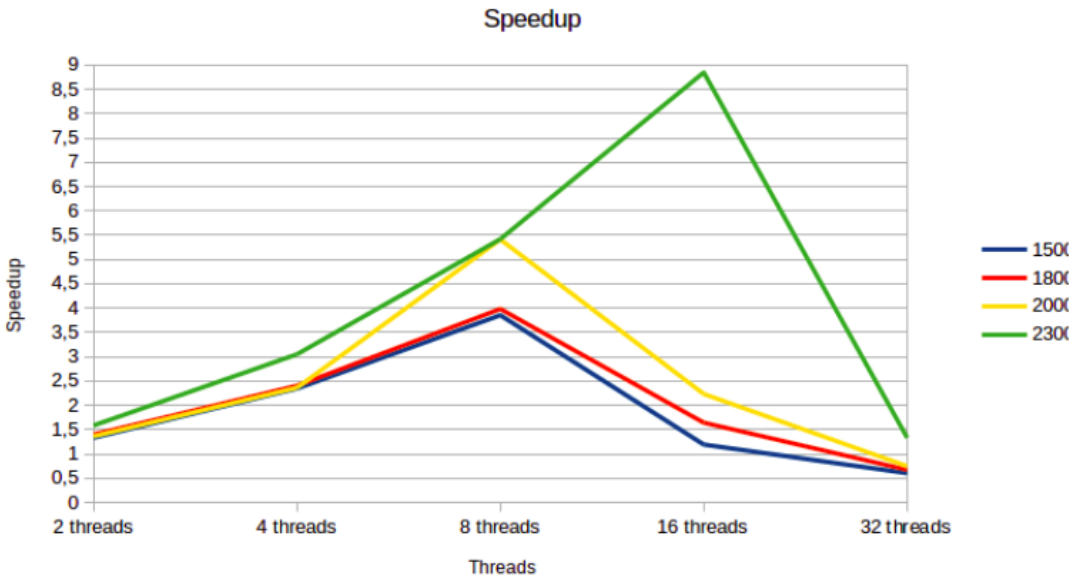

Figura 2 – Speedup da abordagem por elementos

3.2 Eficiências

3.2.1 Por linhas

A eficiência por linhas teve uma inconsistência quando o problema é 2000. Isto era esperado por conta dos speedups adquiridos.

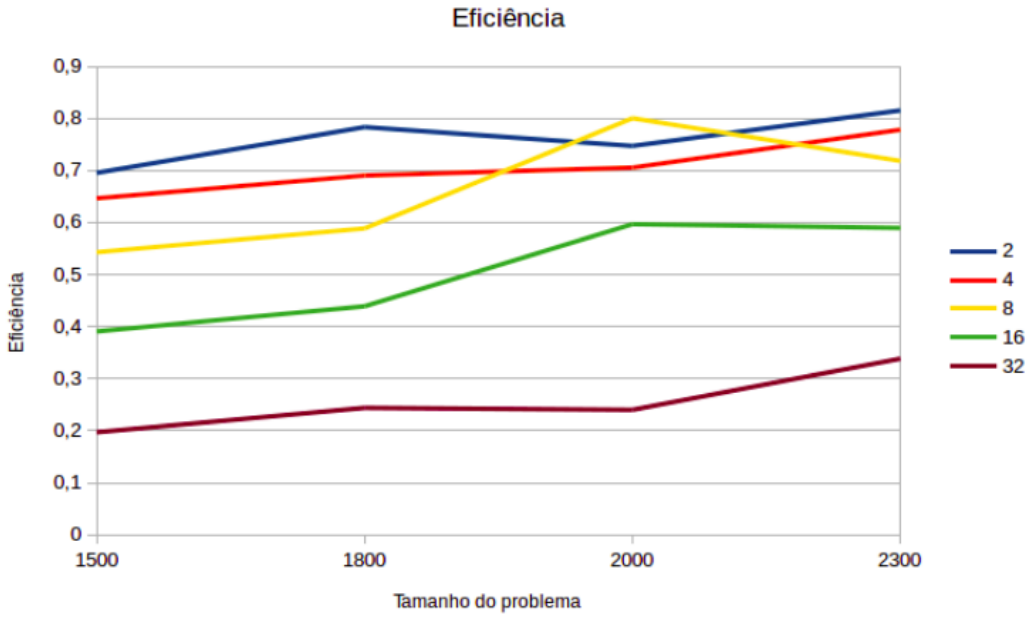

Figura 3 – Eficiência da abordagem por linhas

3.2.2 Por elementos

Também foram obtidos resultados esperados nestas eficiências. É visível que a linha de 32 threads não obteve bons valores. As quebras de runtime dos algoritmos geraram tais resultados.

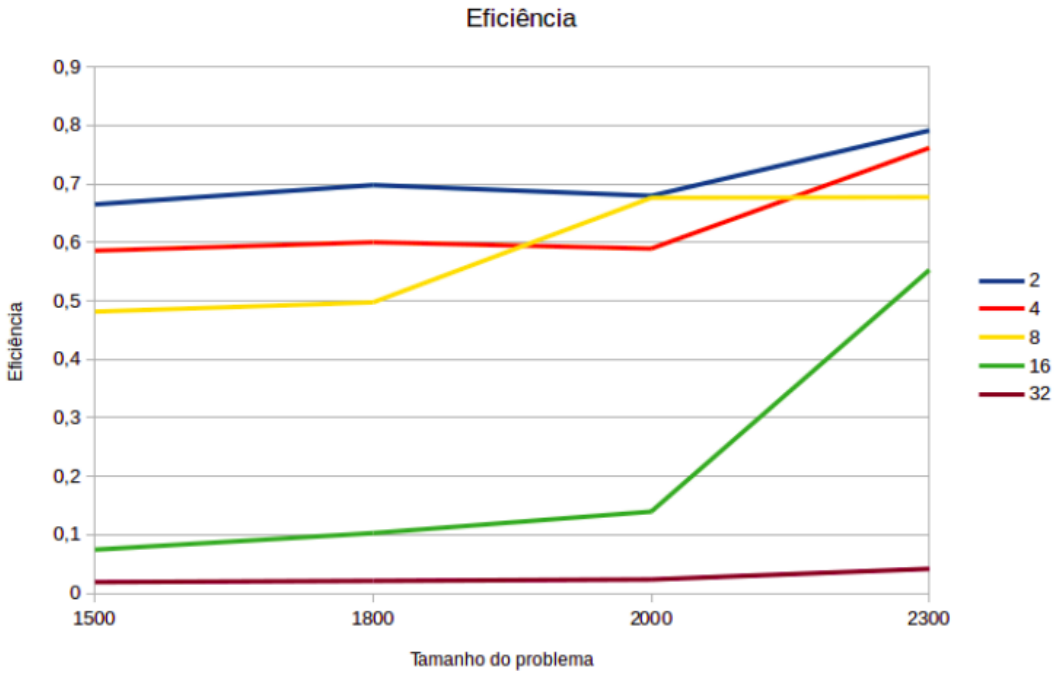

Figura 4 – Eficiência da abordagem por elementos

3.3 CSA

Por não ter sido feito o código serial do CSA, não foi possível criar os gráficos de speedup e eficiência. No entanto, os runtimes da execução no supercomputador foram bons. Os valores sempre caíram ao dobrar as threads, o que indica possíveis bons speedups.

4 Conclusão

As implementações dos algoritmos da multiplicação de matrizes foram relativamentes simples, mas as execuções atrapalharam as análises. Pretendo resolver o problema e conseguir dados mais fiéis.

O CSA foi complicado e duradouro de se implementar. Após vários erros e reformulações de código, consegui o resultado desejado. Porém, por não conseguir finalizar corretamente o mesmo código feito de forma serial, não pude analisar os resultados corretamente. Creio que, encontrando os erros no código serial e executando de forma correta, as análises possam mostrar bons resultados. Os resultados do código paralelo foram bons e animadores.