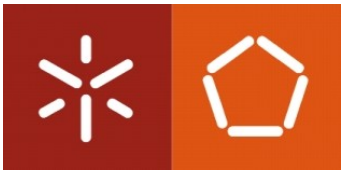

Universidade do Minho
4º do Curso de Mestrado Integrado em Engenharia Informática

Unidade Curricular de Paradigmas de Computação Paralela

Relatório Trabalho Prático 1

Paralelismo em OpenMP ®

Ana Sousa A69855
Carlos Sá A59905

Braga, 1 de Novembro de 2015

Conteúdo

1 Introdução

Este relatório descreve todas as etapas realizadas ao longo deste trabalho prático onde se pretendia que fossem aplicados os conhecimentos adquiridos sobre paralelização em sistemas de memória partilha. De seguida é feita uma apresentação do caso de estudo - Multiplicação de uma matriz esparsa em formato COO por um vetor - e, no capítulo seguinte, é explicada toda a implementação dos algoritmos.

Para concluir este trabalho existe um capítulo onde se fala sobre os testes realizados e onde se faz a análise dos resultados obtidos.

1.1 Caso de Estudo

O caso de estudo é o algoritmo de multiplicação de uma matriz esparsa em formato COO (*Coordinate list*) por um vector.

Uma matriz esparsa (***sparse matrix***) é uma matriz em que o número de elementos cujo valor é 0 é relativamente elevado face ao tamanho da matriz. A percentagem de elementos de uma matriz esparsa que são 0 pode, contudo, variar sem perda de designação. Na secção sobre os dados de entrada veremos isso em mais pormenor.

O primeiro desafio foi implementar o algoritmo sequencial que recebesse uma matriz esparsa A, um vetor x e fizesse o cálculo $y = Ax$ em que y corresponde ao resultado do cálculo efectuado. Com recurso a uma *API* de exploração de paralelismo em modelos de memória partilhada o código sequencial deveria ser paralelizado aplicando um conjunto de fieoptimizações ao código sequencial criado. A API utilizada será o **OpenMP**.

Depois de implementada a versão sequencial e a versão paralela do algoritmo foram realizadas experências que permitissem tirar conclusões relativamente ao comportamento do algoritmo para diferentes dados de entrada (tempos de execução, utilização dos cores disponíveis para o cálculo e a forma como a memória está a ser aproveitada).

O que interessa realçar é que entre os aspectos fundamentais a ter em conta nas matrizes escolhidas para testar o algoritmo estão o tamanho das matrizes esparsas, os tipos dos seus elementos e as várias configurações de valores das matrizes.

2 Dados de Input

Para que seja possível realizar o cálculo $y = Ax$ sendo $\boldsymbol{A}$ uma matriz esparsa e $\boldsymbol{x}$ um vetor, será necessário que o nosso algoritmo receba uma matriz esparsa e o respectivo vector. No âmbito da nossa análise, consideramos sempre que $\boldsymbol{x}$ será sempre um vector não nulo. De acordo com a definição de matriz esparsa encontrada no *Wikipédia*:

> “ *In numerical analysis, a sparse matrix is a matrix in which most of the elements are zero. By contrast, if most of the elements are nonzero, then the matrix is considered dense. The fraction of non-zero elements over the total number of elements (i.e., that can fit into the matrix, say a matrix of dimension of m x n can accommodate m x n total number of elements) in a matrix is called the sparsity (density).* ”
>
> Wikipédia - Sparse Matrix, *2015*

As matrizes a utilizar para testar o nosso algoritmo deverão então ser matrizes com um grande número de zeros em relação ao número total de elementos. Para tal foi necessário definir a percentagem de elementos não zero face ao número total de elementos da matriz, para que trabalhemos sempre com matrizes que são de facto esparsas e não densas. Com base numa pequena pesquisa que fizemos achamos que 25% de elementos não zero seria bom.

2.1 Geração da Matriz Esparsa, Vetor e conversão para formato COO

Para a geração das matrizes esparsas foi feita uma função (*double** geraMatriz (int nLinhas, int nCols, int nnz)*) que com base no número de linhas e colunas, que foram passados como parâmetros no momento da execução do programa, e no número de elementos que não são zero, que foi calculado com base nesses dois valores e tendo em conta a percentagem de não zeros (25%), ou seja, *nnz = (nLinhas * nCols) * 0.25*, gera de forma aleatória a matriz.

As posições (linha, coluna) da matriz e os valores a colocar nessas posições são gerados com o auxílio da função predefinida *rand()* e ajustando os intervalos de geração aos valores que fazem sentido ter. Para o caso dos valores a guardar em cada posição usa-se também uma estratégia para que sejam gerados valores decimais.

Depois de gerada a matriz na sua forma densa ela é convertida para o formato COO. A matriz no formato COO tem de guardar apenas a informação relativa às linhas, colunas e valores não nulos que aparecem na matriz esparsa. Para guardar esta informação optamos por ter um array com tantas linhas quantas a matriz origem tem e colocar em cada linha um array em que na primeira coluna é guardado o comprimento do array e em que nas posições seguintes são guardados os pares (coluna, valor), em posições contíguas e pela ordem dita; ou seja, o array tem tantas colunas como o dobro do total de não zeros existem na linha mais uma posição onde se guarda este tamanho.

Para a geração do vetor o processo foi semelhante. Para cada posição do vetor é feita a geração aleatória de um valor entre [0, 20] e multiplicado esse valor por 0.25 apenas para que os valores sejam decimais. Em vez de o vetor ser guardado tal e qual como na sua forma visual, isto é, em coluna, o vetor é guardado em forma de linha sendo que o número de colunas é igual ao número de linhas do vetor idealizado.

Para que fosse possível usar-se sempre a mesma matriz e vetor nos testes, para permitir que os tempos fossem comparavéis, usou-se a função predefinida *srand(1)* que garante que cada vez que a função `rand()` é executada é gerada a mesma sequência de valores que foi gerada anteriormente.

Tanto para a geração da matriz como para a geração do vetor está a ser feita alocação dinâmica de memória (usando *calloc()*) para armazenar a informação.

Achamos que esta forma de armazenamento das matrizes em formato COO e dos vetores nos iria facilitar a paralelização da multiplicação e que seria uma boa forma de explorarmos a localidade espacial da hierarquia de memória.

Para melhor se perceber a forma como estão armazenados os dados segue-se uma imagem ilustrativa (Figura 1) onde NL é o número total de linhas da matriz e NC é o número total de colunas da matriz.

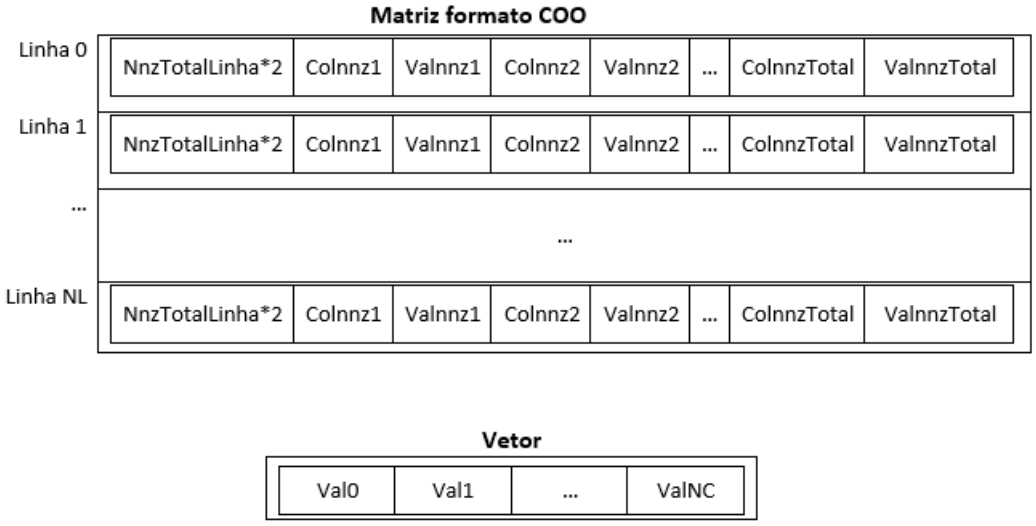

Figura 1: Esquema de armazenamento em memória da matriz em formato COO e do vetor

3 Implementação algoritmo

Nesta seção é feita a explicação da implementação de cada um dos algoritmos criados pelo grupo.

3.1 Algoritmo Sequencial

Depois de gerada a matriz esparsa em formato COO e o vetor tal e qual como explicado na secção anterior (2.1) foi dinamicamente alocada memória para guardar o resultado da multiplicação. Sendo que este resultado é guardado num array que contém tantas linhas quantas as da matriz esparsa.

A multiplicação da matriz pelo vetor é feita percorrendo o array onde se encontram armazenados os dados da matriz esparsa linha a linha. Para cada linha é percorrido o array que contém os pares (coluna, valor) , caso a primeira posição desse array seja maior que zero, e é feita a soma das multiplicações dos valores da matriz pelo valores do vetor respetivos à coluna em que o valor aparece na matriz, isto é,

```
  for(i=0; i<nLinhas; i++) {
  n = (int)coo[i][0];
  soma=0;
  for(j=1; j<=n; j+=2){
   soma += coo[i][j+1]*vect[0][(int)coo[i][j]];
  }
  result[i] = soma; }
```

O código sequencial completo pode ser encontrado no anexo G.

Para não serem efetuados diversos acessos à memória para alterar o valor do resultado de uma linha faz-se o somatório numa variável (*soma*) e só no final é que se coloca o valor em memória. Na mesma perspetiva, visto que a condição $j <= coo[i][0]$ é testada várias vezes, decidiu-se guardar o valor de $coo[i][0]$ numa variável e fazer antes $j <= n$.

Este algoritmo é "amigável"da hierarquia de memória visto que tirar partido da localidade espacial, ou seja, o acesso aos dados é feito pela mesma ordem em que estes estão armazenados, o que aumenta a probabilidade de na próxima iteração do ciclo for interior os dados já estarem em *cache*, diminuindo-se assim os acessos à memória.

3.2 Algoritmo Paralelo

Relativamente ao algoritmo sequencial o que foi feito foi a adicionação de diretivas OpenMP ao código.

Depois de várias tentativas e testes com as diferentes diretivas o resultado final foi:

```
#pragma omp parallel
 {
 #pragma omp for
 for(i=0; i<nLinhas; i++){
  soma=0;
  n = coo[i][0];

  for(j=1; j<=n; j+=2){
   soma += coo[i][j+1]*vect[0][(int)coo[i][j]];
  }
  result[i] = soma;
 }
}
```

O que está a ser feito é a paralelização do ciclo for externo, o que se traduz na atribuição do cálculo da multiplicação de cada linha a uma *thread* diferente, logo apenas uma *thread* acede à posição do array dos

resultados relativa à linha. Desta forma, asseguramos que não existem *data races*. Apesar deste facto, tentamos usar as diretivas *#pragma omp atomic* e *#pragma omp critical* mas, por algum motivo que não conseguimos identificar, por vezes ocorria um *Segmentation Fault*.

Tentamos também a diretiva *#pragma omp collapse(2)*, fazendo previamente uma reorganização do código atual, mas existiam *data races* visto que ao se juntarem os dois ciclos for existiriam várias *threads* a aceder à mesma posição de memória.

O código paralelo completo pode ser encontrado no anexo H.

4 Versão Sequencial VS Versão Paralela

Ao longo do trabalho fomo-nos apercebendo que a multiplicação de matrizes por vetores com um baixo número de elementos, geralmente não trazem ganhos significativos. Inicialmente a nossa versão paralela do algoritmo até tinha tempos de execução maiores que os tempos da versão sequencial. A razão que encontramos para que tal aconteça está relacionado com a criação das threads. Para matrizes de pequena dimensão, é gasto mais tempo na criação da *team* e a escalonar trabalho para as threads da team do que a executar o código do programa, o que inevitavelmente se traduz diretamente no tempo de execução.

4.1 Testes no MacBook Air Early 2014

Em anexo, temos o resultado das diferentes medições realizadas.

No anexo B podem ser consultados os dados da recolha dos tempos que foram feitos na máquina do grupo (MacBook Air). Podemos verificar que, para a matriz mais pequena de todas, é onde existem ganhos maiores. Ainda existe algum overhead criado pela criação das threads e escalonamento de trabalho para estas. Note-se que todos os ganhos se referem à comparação entre TexecSeq e TexecParallel para os diferentes números de Cores/Threads disponíveis. Para ajudar a análise, poderá visualizar os gráficos de SpeedUp's no anexo C para os vários tamanhos de matriz.

Existe um comportamento semelhante apartir do momento em que se executa o algoritmo em paralelo para os diferentes tamanhos de matrizes. Há uma tendência de perda de performance. De facto a máquina em causa, sendo dual core, apenas possui 2 cores independentes, tendo no então mais dois "cores"simulados. No entanto esses dois "cores"(threads) não possuem unidades funcionais independentes pelo que as unidades funcionais dos 2 cores dedicados são divididos com as threads para o processamento das iterações do ciclo. Então existe uma perda progressiva do ganho apartir das 4 threads. No anexo F poderá visualizar uma pequena comparação dos ganhos de performance para a multiplicação das diferentes matrizes pelo vetor na nossa máquina e no SeARCH.

4.2 Testes no SeARCH - compute-652-2

Após realizarmos o teste do algoritmo na nossa máquina resolvemos fazer uma análise comparativa com os resultados obtidos no SeARCH. A recolha dos tempos de execução no SeARCH estão no anexo D. Para cada cálculo da multiplicação foi feito um conjunto de 10 medições no search (tal como no computador pessoal) para cada número de cores, para cada um dos tamanhos de matrizes e vetores. Foi feita a mediana de cada conjunto de 10 medições para calcular o ganho correspondente. Claro que a nível de ganhos, estes foram superiores no SeARCH em comparação com o Macbook Air (mais explicito na matriz 8192x8192) porque as próprias caracteristicas do hardware o permitem. Pela análise dos gráficos de SpeedUp's do MacBook Air e do SeARCH concluímos que é possível obter ganhos maiores no SeARCH para matrizes sucessivamente maiores a nível de tamanho de input. O SeARCH consegue lidar com matrizes de maior dimensão melhor que o MacBook Air para o mesmo tamanho de input. O SeARCH dispõe de um maior número de cores dedicados, possui caches maiores, então o nosso algoritmo consegue tirar mais partido desta hierarquia de memória uma vez que os dados são percorridos por linhas, acedendo a posições consecutivas de memória. Consegue-se tirar um maior partido da localidade espacial. Na nossa matriz COO condensamos os valores que pertencem todos a uma mesma linha da matriz esparsa, num array. Assim, cada linha apenas é percorrida uma vez e todos os valores dessa linha podem ser processados e acedidos em bloco. Os gráficos de SpeedUps do SeARCH encontram-se no anexo E.

5 Testes

Nesta seção são explicados todos os detalhes relativos aos testes realizados, bem como é feita uma análise dos resultados obtidos depois de corridos os testes.

5.1 Compilação dos algoritmos

Ao longo das aulas vimos que o compilador da *GNU* para C (gcc) inclui alguns níveis de optimização que aplicam um conjunto de técnicas automáticas que permitem optimizar código sem que o programador nada tenha que fazer, sendo necessário apenas o uso de flags para indicar a "profundidade"dessas optimizações. Para definir esse nível de otimização usamos a flag -O3".

A compilação foi feita usando **gcc 4.9.3** no *SeARCH Cluster* e **gcc 5.2.0** no computador pessoal dos alunos:

```
1  $ gcc -O3 -Wall -Wextra -std=c99 -fopenmp -o tp1_<seq><paralel> tp1_<seq↩
       ><paralel>.c
```

Depois do código compilado para ambas as implemetações foram feitos os testes que permitem fazer comparações entre os dois algoritmos implementados.

5.2 Tamanho de Input

Por forma a conseguir avaliar a performance do algoritmo foi necessário definir o tamanho dos dados de entrada. Esse tamanho deve ser um tamanho relativamente razoável para conseguirmos maximizar o número de medições que nos é possível fazer mas, ao mesmo tempo, que necessite algum esforço de computação para que seja possível a avaliação do comportamento da memória e tirar conclusões sobre o tempo de execução.

Dada a hierarquia de memória das máquinas, achamos pertinente realizar testes com tamanhos que permitam perceber o impacto das penalizações nos acessos à memória. Assim sendo, como será possível perceber por este relatório os tamanhos de input (tamanho das matrizes de input vector) ultrapassam o tamanho da cache dos processadores. O tamanho da cache do Macbook Air Early 2014 não tem sequer 2^2 MB de cache. O SeARCH dispõe de cerca de 25MB de Cache nos processadores Xeon, o que também obrigará a a miss's na cache para os nossos volumes de matrizes.

Os elementos das matrizes e dos vectores são **doubles**. Assim, as nossas matrizes e vetores são de tamanho 2^{11}, 2^{12} e 2^{13}. Logo, temos matrizes para testar o algoritmo de tamanhos $2048 \times 2048, 4096 \times 4096 e 8192 \times 8192$.

5.3 Metodologia e Condições de medição

A métrica utilizada para avaliar o desempenho do nosso algoritmo (quer o sequencial, quer o paralelo) será o **Tempo de execução**. É com base nela que iremos explorar os ganhos obtidos com as optimizações introduzidas no algoritmo.

Ao nível da nossa máquina pessoal, houve um conjunto de precauções que foram tomadas por forma (tentar) minimizar factores externos que provoquem alteração dos tempos de execução medidos. Como tal, na nossa máquina pessoal:

- Usamos o contador de tempos do OpenMP (com omp_get_wtime());
- A resolução de tempos utilizada é o **milissegundo (ms)**;
- Desligamos todos os possíveis factores de intrusão como cargas de outras aplicações e processos desnecessários;
- Utilizamos a máquina sempre ligada à corrente para evitar oscilações da frequência do CPU;
- As áreas delimitadas para a medição dos tempos não inclui nenhum tipo de I/O (como *printf's*, por exemplo);

- As medições recolhidas foram aquelas que resultaram de tempos após aquecimento da cache.
- Recolhemos cerca de 10 medições para cada teste.
- Ao total das medições realizadas em cada teste, foi feita a **mediana** desses valores para atenuar as maiores oscilações de valores;

5.3.1 Factores externos

Existe um conjunto de outros factores externos que não conseguimos controlar. Nas nossas máquinas, há processos que podem estar a consumir CPU e que podem interferir com os tempos. Esta questão também não foi possível de ser evitada no *SeARCH*. Idealmente, deveriamos ter reservado uma máquina do SeARCH de 48 cores para tentar fazer a medição dos tempos. Desta forma, as medições seriam possivelmente mais fidedignas pois teriamos uma máquina completa dedicada a executar o nosso código sem "interferências"de outros trabalhos. A submissão Job para uma máquina de 48 cores no SeARCH foi feita nesse sentido mas infelizmente, o tempo de espera era demasiado elevado. Assim sendo, todos os *jobs* submetidos no SeARCH foram feitos com pedidos de 24 Cores.

6 Conclusão

Globalmente o grupo acredita ter cumprido com os diferentes objetivos para este trabalho. Graças a ele aprendemos a utilizar diretivas poderosas que nos permitem tirar um maior proveito hardware disponível e melhorar os nossos algoritmos.

Por forma a perceber melhor como funciona o paralelismo e podermos chegar a este resultado final, foi necessário estudar várias optimizações disponíveis ao programador como o desdobramento de ciclos, as várias diretivas OpenMp e perceber como funciona o paradigma de memória partilhada. Percebemos concretamente que nem todos os blocos de código se paralelizam trivialmente e o programador é o primeiro responsável pela correcção de um algoritmo quando se pretende paralelizar um código num paradigma de memória partilhada com diretivas OpenMP. Várias foram as vezes que nos confrontamos com data races, quando tentamos modificar o algoritmo para collapsar os dois ciclos for com a directiva *pragma omp collapse*, entre outras diretivas que fomos utilizando sem que nem sempre produzissem o resultado correto.

Como trabalho futuro, achamos que seria um bom exercicio voltar a testar o algoritmo mas ver como este se comporta para matrizes esparsas de densidade variável.

A
Máquinas de teste

Para a realização dos testes foram usadas duas máquinas diferentes, o *SeARCH Cluster* e o computador pessoal de um dos elementos do grupo. Listam-se a seguir as caraterísticas delas.

HW	**SeARCH (compute-652-2)**	**MacBook Air Early 2014**
Processador	Intel(R) Xeon(R) E5-2670 v2 @ 2.50GHz	Intel Core i5 Dual-Core 1,4 GHz (até 2,7 GHz)
Cache:	25MB	32KB (L1) + 256 KB (L2) + 3MB (L3 shared)
Memória	64GB	8GB LPDDR3 1600 MHz
Sistema Operativo	Rocks 6.1 (Emerald Boa)	OS X El Capitan 10.11.1 (15B42)

B
Recolha de Tempos MacBook Air Early 2014

Tam. Matriz	#Threads	Mediana dos tempos(ms)	Ganho Seq Vs Paralelo
2048*2048	0(seq)	1,906037331	0
	2	1,55043602	1,229355682
	4	1,334428787	1,428354476
	6	1,372456551	1,388777903
	8	1,459002495	1,306397582
4096x4096	0(seq)	8,863091469	0
	2	5,028963089	1,76240933
	4	4,70995903	1,881776765
	6	4,541516304	1,95157099
	8	4,745602608	1,867642995
8192x8192	0(seq)	32,81247616	0
	2	18,72432232	1,752398597
	4	17,84455776	1,838794584
	6	17,71593094	1,852145183
	8	17,49300957	1,875747911

C
SpeedUp's - Macbook Air Early 2014

C.1 Multiplicação 2048x2048 * 2048

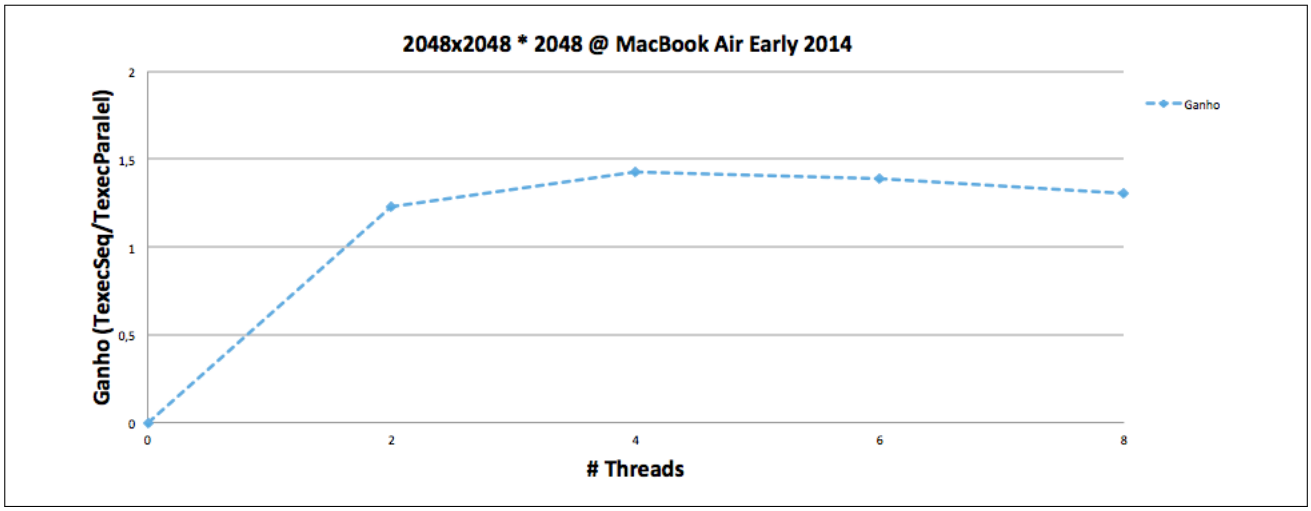

Figura 2: Multiplicação matriz 2048x2048 por vetor de tamanho 2048

C.2 Multiplicação 4096x4096 * 4096

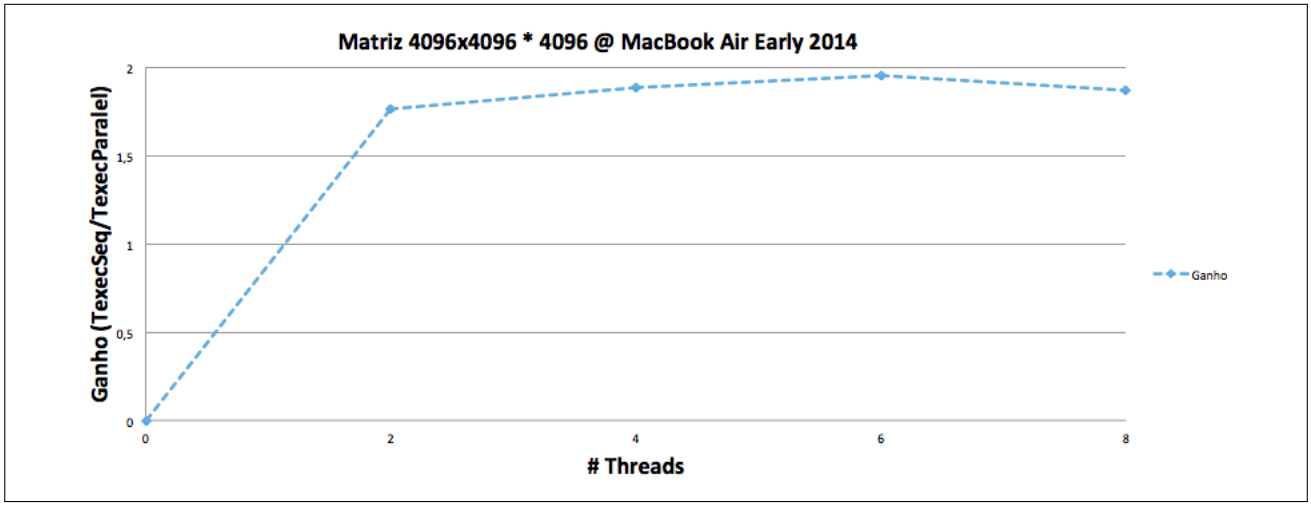

Figura 3: Multiplicação matriz 4096x4096 por vetor de tamanho 4096

C.3 Multiplicação 8192x8192 * 8192

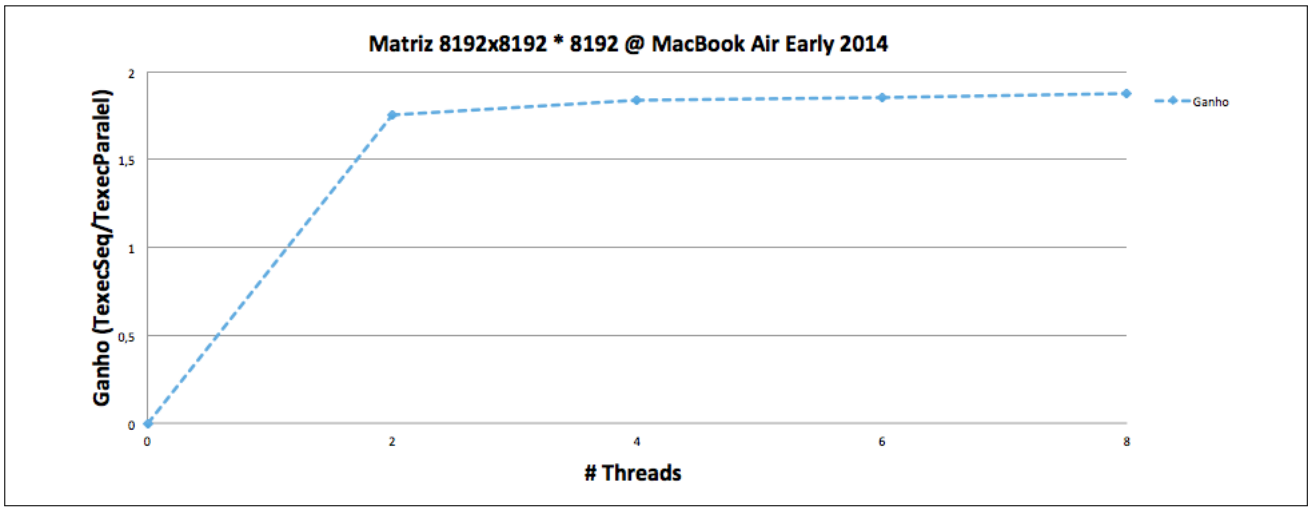

Figura 4: Multiplicação matriz 8192x8192 por vetor de tamanho 8192

D

Recolha de Tempos SeARCH - compute-652-2

Tam. Matriz	#Threads	Mediana dos tempos(ms)	Ganho Seq Vs Paralelo
2048*2048	0(seq)	1,986813964	0
	2	2,371700481	0,837717064
	4	1,409262419	1,409825408
	6	1,095122891	1,814238367
	8	0,984443817	2,018209602
	10	0,936415279	2,121723137
	12	0,956980046	2,076128936
	14	0,905001536	2,19537082
	16	0,955306692	2,079765567
	18	0,98165893	2,0239351
	20	1,044455683	1,902248221
	22	1,096804626	1,811456586
	24	1,258009346	1,57933164
4096*4096	0(seq)	8,630240569	0
	2	7,962655509	1,0838395
	4	4,74948599	1,817089383
	6	3,789767623	2,277247955
	8	3,202710068	2,694668074
	10	2,996010007	2,880578018
	12	2,71175825	3,18252579
	14	2,621314023	3,292333728
	16	2,460590098	3,507386531
	18	2,729684114	3,16162611
	20	3,042481607	2,836579373
	22	3,091085469	2,791977334
	24	3,228514455	2,673130534
8192x8192	0(seq)	39,83017476	0
	2	24,23084131	1,6437801
	4	14,29473422	2,786352942
	6	10,09257766	3,946481869
	8	8,62235087	4,619410108
	10	8,379518986	4,753276988
	12	7,935790345	5,019055826
	14	7,737488486	5,147687759
	16	9,130208986	4,362460358
	18	9,425137658	4,225951515
	20	9,116475005	4,36903241
	22	10,34303289	3,850918313
	24	10,61496534	3,752266114

E
SpeedUp's - SeARCH - compute-652-2

E.1 Multiplicação 2048x2048 * 2048

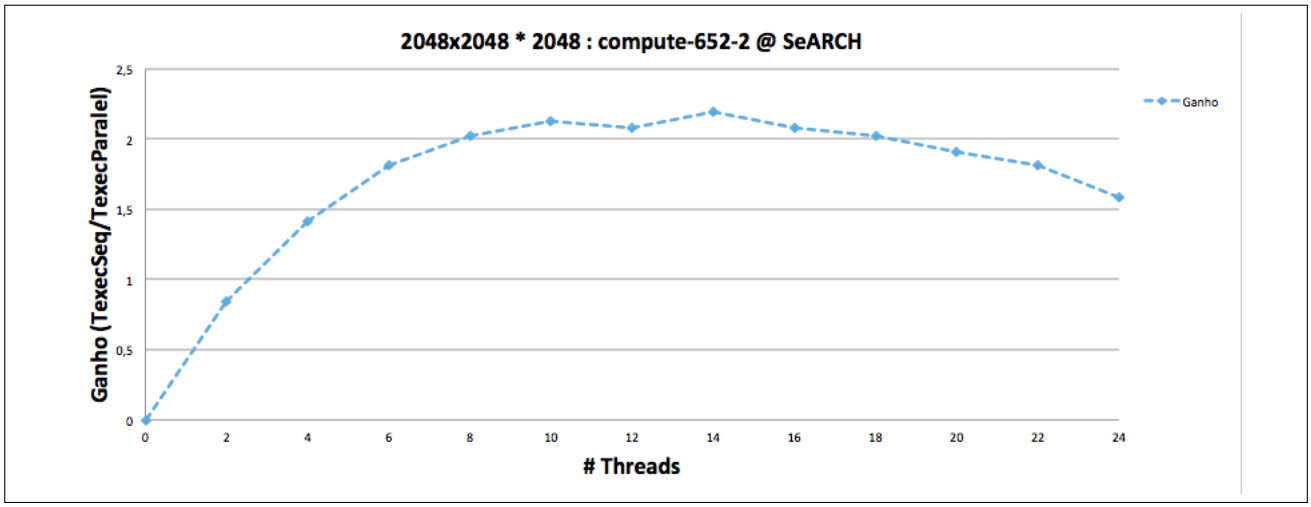

Figura 5: Multiplicação matriz 2048x2048 por vetor de tamanho 2048

E.2 Multiplicação 4096x4096 * 4096

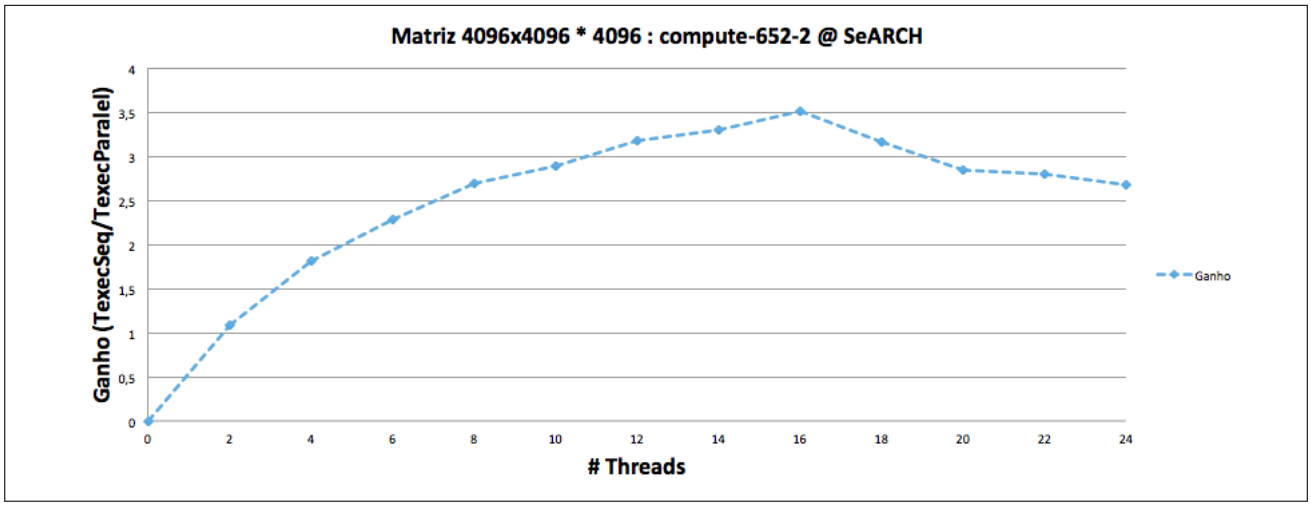

Figura 6: Multiplicação matriz 4096x4096 por vetor de tamanho 4096

E.3 Multiplicação 8192x8192 * 8192

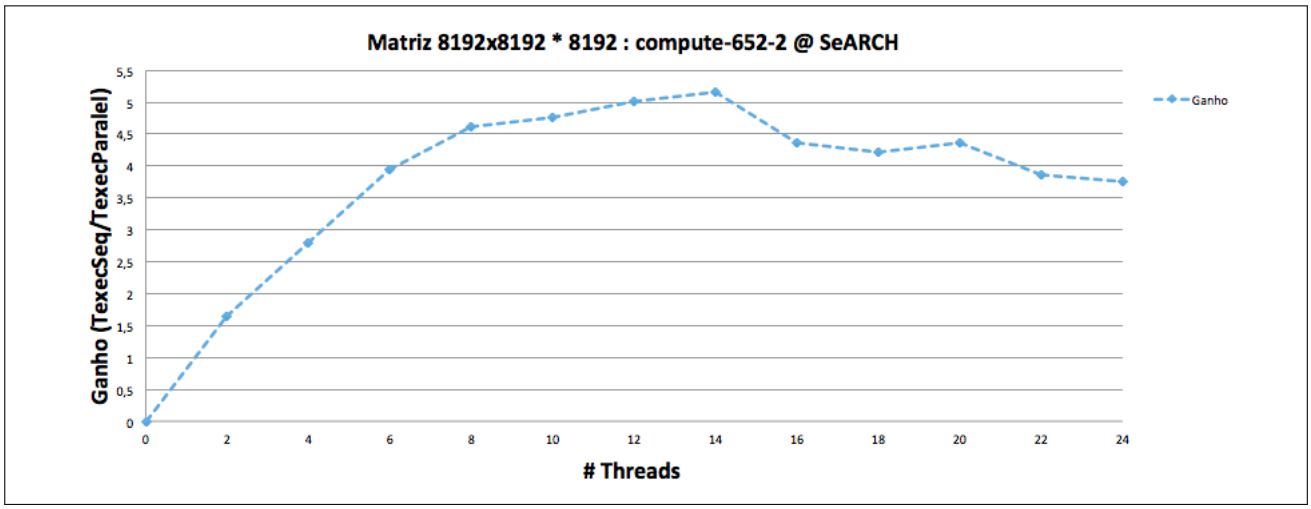

Figura 7: Multiplicação matriz 8192x8192 por vetor de tamanho 8192

F Comparação SpeedUp's

F.1 MacBook Air Early 2014

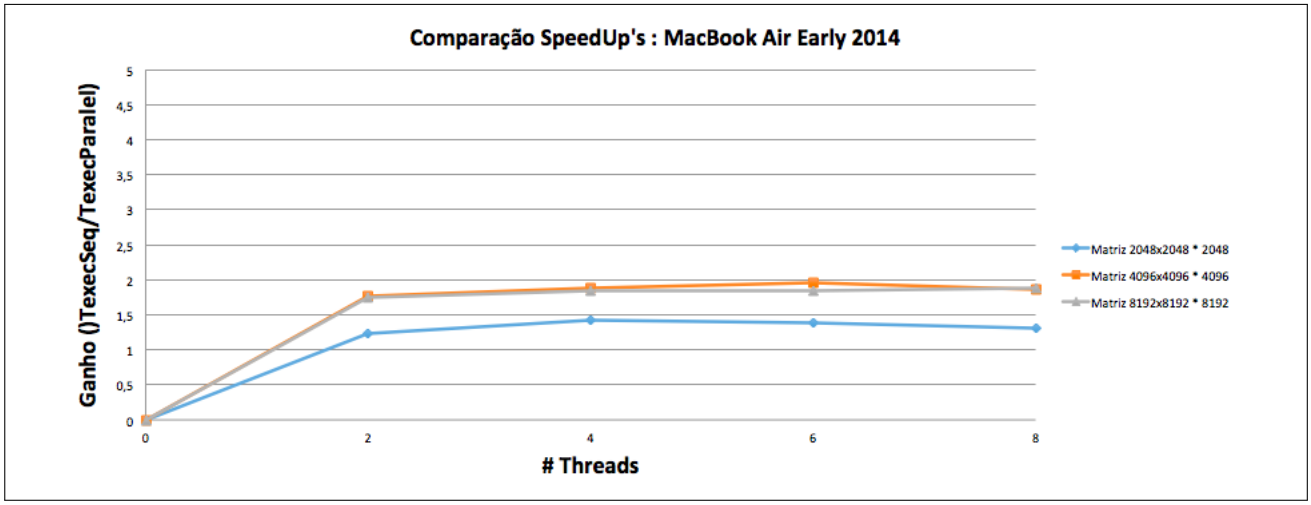

Figura 8: Comparação dos diferentes SpeedUp's para vários tamanhos de matriz e vetor no MacBook Air Early 2014

F.2 SeARCH - compute-652-2

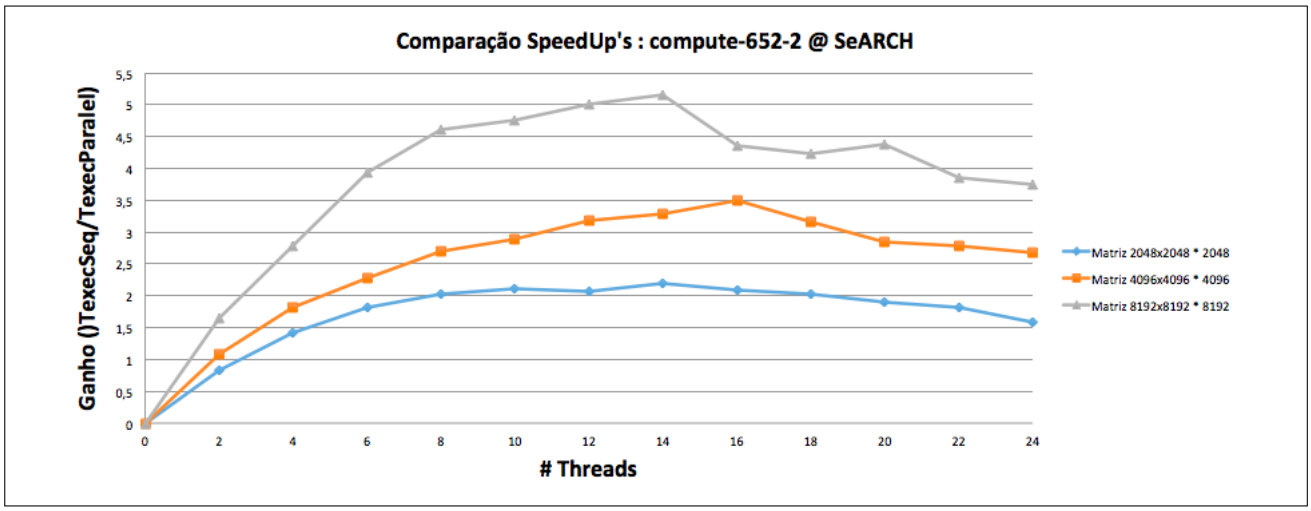

Figura 9: Comparação dos diferentes SpeedUp's para vários tamanhos de matriz e vetor no SeARCH - compute-652-2

G Código sequencial

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <omp.h>

double** geraMatriz (int nLinhas, int nCols, int nnz) {
 int i, j, x, y, aux, cont;
 double **mat = (double **) calloc (nLinhas, sizeof(double*));
 double **coo = (double **) calloc (nLinhas, sizeof(double*));
 double valRand;

 for (i=0; i<nLinhas; i++) {
  mat[i] = (double*) calloc (nCols, sizeof(double));
 }

 srand(1);
 for (i=0; i<nnz; i += aux) {
  x = rand() % nLinhas;
  y = rand() % nCols;
  if (mat[x][y] == 0.0) {
   while ((valRand = (rand() % 50) * 0.25) == 0) {;}
   mat[x][y] = valRand;
   aux = 1;
  } else {
   aux = 0;
  }
 }

 for (i=0; i<nLinhas; i++) {
  cont = 0;

  for(j=0; j<nCols; j++) { if (mat[i][j] != 0.0) {cont++;} }
  if (cont>0) {
   coo[i] = (double*) calloc ((cont*2)+1, sizeof(double));
   coo[i][0] = cont * 2;
   aux = 1;
   for (j=0; j<nCols; j++) {
    if (mat[i][j] != 0.0) {
     coo[i][aux] = j;
     coo[i][aux+1] = mat[i][j];
     aux += 2;

    }
   }
  } else {
   coo[i] = (double*) calloc (1, sizeof(double));
   coo[i][0] = 0;
  }
  free(mat[i]);
 }
 free(mat);
```

```
 return coo;
}

double** geraVetor (int nCols) {
 int i;
 double **vect = (double**) calloc (1, sizeof(double*));
 vect[0] = (double*) calloc (nCols, sizeof(double));
 srand(1);
 for(i=0; i<nCols; i++) {
  vect[0][i] = (rand() % 20) * 0.25;
 }

 return vect;
}

int main(int argc, char *argv[]) {
 unsigned int i, j, n, nnz, nLinhas, nCols, nLinhas_vect;
 double startTime, finalTime;
 double **coo, **vect, *result;
 double soma;

 /* Preenchimento Matriz */
 nLinhas = atoi(argv[1]);
 nCols = atoi(argv[2]);
 nnz = (nLinhas * nCols) * 0.25;
 coo = geraMatriz(nLinhas, nCols, nnz);

 /* Preenchimento Vector */
 nLinhas_vect = atoi(argv[3]);
 vect = geraVetor(nLinhas_vect);

 result = (double*) calloc (nLinhas, sizeof(double));

 /* Multiplicacao */
 startTime = omp_get_wtime();
 for(i=0; i<nLinhas; i++) {
  n = (int)coo[i][0];
  soma=0;
  for(j=1; j<=n; j+=2){
   soma += coo[i][j+1]*vect[0][(int)coo[i][j]];
  }
  result[i] = soma;
 }
 finalTime = omp_get_wtime();

 /* Visualizacao dos resultados */
/* printf("----------Matriz formato COO----------\n");
 for(i=0; i<nLinhas; i++){
  if (coo[i]) {
   n = coo[i][0];
   for(j=1; j<=n; j+=2) {
    printf("%f %f %f\n", (double)i, (double)coo[i][j], (double)coo[i][j+1]);
   }
  }
 }
```

```
 printf("\n----------Vetor----------\n");
 for(i=0; i<nLinhas_vect; i++) {
  printf("%f\n", vect[0][i]);
 }

 printf("------------Resultado Multiplicacao----------\n");
 for(i=0; i<nLinhas; i++)
  printf("%f\n", result[i]);
*/
 printf("Time seq: %.12f\n",(finalTime - startTime)*1000);

 /* Libertar a memoria alocada */
 for(i=0; i<nLinhas; i++) {
  free(coo[i]);
 }
 free(coo);
 free(vect);
 free(result);

 return 0;
}
```

H Código paralelizado

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <omp.h>

double** geraMatriz (int nLinhas, int nCols, int nnz) {
 int i, j, x, y, aux, cont;
 double **mat = (double **) calloc (nLinhas, sizeof(double*));
 double **coo = (double **) calloc (nLinhas, sizeof(double*));
 double valRand;

 for (i=0; i<nLinhas; i++) {
  mat[i] = (double*) calloc (nCols, sizeof(double));
 }

 srand(1);
 for (i=0; i<nnz; i += aux) {
  x = rand() % nLinhas;
  y = rand() % nCols;
  if (mat[x][y] == 0.0) {
   while ((valRand = (rand() % 50) * 0.25) == 0) {;}
   mat[x][y] = valRand;
   aux = 1;
  } else {
   aux = 0;
  }
 }

 for (i=0; i<nLinhas; i++) {
  cont = 0;

  for(j=0; j<nCols; j++) { if (mat[i][j] != 0.0) {cont++;} }
  if (cont>0) {
   coo[i] = (double*) calloc ((cont*2)+1, sizeof(double));
   coo[i][0] = cont * 2;
   aux = 1;
   for (j=0; j<nCols; j++) {
    if (mat[i][j] != 0.0) {
     coo[i][aux] = j;
     coo[i][aux+1] = mat[i][j];
     aux += 2;
    }
   }
  } else {
   coo[i] = (double*) calloc (1, sizeof(double));
   coo[i][0] = 0;
  }
  free(mat[i]);
 }
 free(mat);

 return coo;
```

```
}

double** geraVetor (int nCols) {
 int i;
 double **vect = (double**) calloc (1, sizeof(double*));
 vect[0] = (double*) calloc (nCols, sizeof(double));
 srand(1);
 for(i=0; i<nCols; i++) {
  vect[0][i] = (rand() % 20) * 0.25;
 }

 return vect;
}

int main(int argc, char *argv[]) {
 unsigned int i, j, n, nnz, nLinhas, nCols, nLinhas_vect;
 double startTime, finalTime;
 double **coo, **vect, *result;
 double soma;

 /* Preenchimento Matriz */
 nLinhas = atoi(argv[1]);
 nCols = atoi(argv[2]);
 nnz = (nLinhas * nCols) * 0.25;
 coo = geraMatriz(nLinhas, nCols, nnz);

 /* Preenchimento Vector */
 nLinhas_vect = atoi(argv[3]);
 vect = geraVetor(nLinhas_vect);

 /* Multiplicacao */
 result = (double*) calloc (nLinhas, sizeof(double));
 omp_set_num_threads(atoi(argv[4]));

 startTime = omp_get_wtime();
 #pragma omp parallel
 {
 #pragma omp for
 for(i=0; i<nLinhas; i++){
  //printf("Ciclo_Thread:%d iteração:%d paralelo:%d\n", omp_get_thread_num(),i, ↩
      omp_in_parallel());
  soma=0;
  n = coo[i][0];

  for(j=1; j<=n; j+=2){
   soma += coo[i][j+1]*vect[0][(int)coo[i][j]];
  }
  result[i] = soma;
 }
 }
 finalTime = omp_get_wtime();

 //printf("Fim do ciclo_Paralelo? %d\n", omp_in_parallel());

 /* Visualizacao dos resultados */
/* printf("----------Matriz formato COO----------\n");
```

```
 for(i=0; i<nLinhas; i++){
  if (coo[i]) {
   n = coo[i][0];
   for(j=1; j<=n; j+=2) {
    printf("%f %f %f\n", (double)i,(double)coo[i][j], (double)coo[i][j+1]);
   }
  }
 }

 printf("\n----------Vetor----------\n");
 for(i=0; i<nLinhas_vect; i++) {
  printf("%f\n", vect[0][i]);
 }

 printf("----------Resultado Multiplicacao----------\n");
 for(i=0; i<nLinhas; i++)
  printf("%f\n", result[i]);
*/
 printf("Time parallel: %.12f\n",(finalTime - startTime)*1000);

 /* Libertar a memoria alocada*/
 for(i=0; i<nLinhas; i++) {
  free(coo[i]);
 }
 free(coo);
 free(vect[0]);
 free(vect);
 free(result);

 return 0;
}
```