Universidade de São Paulo
Instituto de Matemática e Estatística
MAC 5742 - Computação Paralela e Distribuída

Exercício Programa 1: OpenMP

Autores:

Diana Naranjo

Walter Perez

São Paulo

Abril 2015

Resumo

Nesse Exercício Programa (EP) o objetivo foi explorar a computação paralela com memoria compartilhada, para isso foi usado o padrão OpenMP. A primeira parte do trabalho explora o cuidado que deve-se ter no momento de realizar o desenvolvimento de programas usando as diretivas do OpenMP. É muito simples cometer erros quando ainda se está pensando de maneira sequencial, ao assumir algum comportamento ou quando não se conhece bem o comportamento padrão das diretivas usadas. A segunda parte do EP procura avaliar as melhoras (ou falta delas) no tempo de execução de um programa alvo, mult.c, que realiza a multiplicação de 2 matrizes. Para avaliar o desempenho da versão sequencial versus a paralela uma serie de experimentos foram realizados. Alguns deles involucraram a alteração do programa para criar distintas zonas paralelas e também a execução deles usando diferentes números de threads. A continuação presentamos os experimentos, resultados e conclusões.

Sumário

1 Introdução

A computação paralela consiste em executar um conjunto de cálculos de maneira simultânea [3]. A meta de este tipo de computação é diminuir o tempo de execução que exigem algumas aplicações, e.g. aquelas com fortes requerimentos de computo. Atualmente, existem computadores com múltiplos cores; além de isso também contam com tecnologias como hyper-threading, o que gera que o sistema operativo assuma que conta com uma maior quantidade de recursos de computo. Os recursos, por tanto, estão prontos para ser usados. Porém, para realmente obter uma melhoria no rendimento não é suficiente executar as aplicações em uma máquina multicore, é necessário que elas troquem seus algoritmos sequências por uma versão paralela. De outra maneira, não estaria-se fazendo uso do processamento paralelo e seus benefícios.

O grande problema é que a troca de algoritmos sequências por paralelos não é um trabalho simples. Existem diversos problemas na geração de códigos paralelos de bom rendimento, alguns deles são [5]:

- Gargalos de comunicação. Debe ter-se em consideração a quantidade processadores que serão utilizados na execução e o overhead produto da comunicação entre estes processadores e a memoria [2]. Isto é importante pois pode acontecer que um programa em versão paralela tome mais tempo na execução do que na versão sequencial. O que se debe a que a memoria só pode ser acessada por um thread em qualquer momento e o custo de manter as caches de cada processador coerente também gera um overhead.
- Balanceamento de carga. Ao fazer uma divisão de trabalho o ideal é que cada uma das partes tenha uma carga igual. Se uma delas tem mais trabalho do que as outras então no final o problema volta a ser sequencial pois a maioria termina com sua carga e umas poucas continuam com o resto do trabalho.
- Problemas na construção do código. Os desenvolvedores estão acostumados a pensar em forma sequencial e por tanto é muito fácil cometer erros de concorrência. As operações simples, como por exemplo a assinação de valores a uma variável, causam resultados inesperados na versão paralela. Isto é devido as condições de corrida, i.e. quando o programa se desenvolve em um ordem diferente ao que o programador planeio.

Para evitar os erros produzidos por o pouco cuidado do desenvolvedor e para maximizar o paralelismo dos algoritmos que são sequências existem conjuntos de diretivas que criam um código paralelo executável. OpenMP é um conjunto de diretivas para o compilar, rutinas de libraria e variáveis de entorno que podem ser usadas para a geração de paralelismo em códigos Fortran e C/C++ [1]. O objetivo é garantir o correto funcionamento dos programas e obter os benefícios do paralelismo.

Neste trabalho exploramos os problemas persistentes no uso das diretivas do OpenMP. Tanto no desenvolvimento de código errôneo, como na geração de overhead pela geração de áreas paralelas ineficientes. Para isso desenvolvemos experimentos para obter o tempo promédio na execução de códigos gerados a partir de distintas áreas paralelas, em ambientes diferentes e com distintas quantidades de threads.

2 Exercício 1

Escolher um código na internet que use as diretivas de compilação do OpenMP, esse código deve ser procurado nos respetivos tutoriais e manuais desse padrão de programação multiprocessamento. A ideia e encontrar erros nessas implementações fornecidas ou apresentadas nos tutoriais consultados. Apresente o código, aponte os problemas e descreva quais são as correções feitas para tirar o erro da aplicação.

2.1 Código com erro

O seguinte programa foi extraído de [6] e tem como objetivo imprimir os valores de $A[i] = i * i$ no mesmo ordem da iteração, neste caso para os valores de i a partir de 0 a 15.

```
1  #include <stdio.h>
2  #include <omp.h>
3  #define SIZE 16
4
5  int main(){
6    int A[ SIZE ] , i ;
7    #pragma omp parallel for schedule( static , 2 ) num_threads( 4 )
         ordered
8    for( i = 0 ; i < SIZE ; i++){
9      A[ i ] = i * i ;
10     printf( "Th[ %d ]: %02d = %03d\n" , omp_get_thread_num() , i , A[ i ]
           ) ;
11   }
12   return 0 ;
13 }
```

O resultado sería:

```
Th[ 0 ]: 00 = 000
Th[ 0 ]: 01 = 001
Th[ 1 ]: 02 = 004
Th[ 1 ]: 03 = 009
Th[ 2 ]: 04 = 016
Th[ 2 ]: 05 = 025
Th[ 3 ]: 06 = 036
```

```
Th[ 3 ]: 07 = 049
Th[ 0 ]: 08 = 064
Th[ 0 ]: 09 = 081
Th[ 1 ]: 10 = 100
Th[ 1 ]: 11 = 121
Th[ 2 ]: 12 = 144
Th[ 2 ]: 13 = 169
Th[ 3 ]: 14 = 196
Th[ 3 ]: 15 = 225
```

2.2 Problemas encontrados

O erro no programa anterior está em usar a **diretiva ordered** sem colocar um **bloco ordered** dentro de for (linhas 9-10) [4]. Por isso que ao executar o programa, o resultado será similar a:

```
Th[ 2 ]: 04 = 016
Th[ 0 ]: 00 = 000
Th[ 3 ]: 06 = 036
Th[ 1 ]: 02 = 004
Th[ 2 ]: 05 = 025
Th[ 0 ]: 01 = 001
Th[ 3 ]: 07 = 049
Th[ 1 ]: 03 = 009
Th[ 0 ]: 08 = 064
Th[ 1 ]: 10 = 100
Th[ 0 ]: 09 = 081
Th[ 2 ]: 12 = 144
Th[ 3 ]: 14 = 196
Th[ 1 ]: 11 = 121
Th[ 2 ]: 13 = 169
Th[ 3 ]: 15 = 225
```

2.3 Correções

Para que o resultado do programa seja correcto deve ser adicionado o bloco ordered e neste caso só é necessário colocá-lo para a linha 10 porque você quer imprimir em ordem, mas também poderia ser colocado para as linhas 9 e 10.

```
1 #include <stdio.h>
2 #include <omp.h>
3 #define SIZE 16
4
```

```
5  int main(){
6    int A[ SIZE ] , i ;
7    #pragma omp parallel for schedule( static , 2 ) num_threads( 4 )
         ordered
8    for( i = 0 ; i < SIZE ; i++){
9      A[ i ] = i * i ;
10     #pragma omp ordered
11     {
12       printf( "Th[ %d ]: %02d = %03d\n" , omp_get_thread_num() , i , A[ i
             ] ) ;
13     }
14   }
15   return 0 ;
16 }
```

2.4 Conclusões

A diretiva ordered deve ser sempre usada com seu bloco ordered para que o compilador pode saber quais tarefas devem ser executadas em ordem, caso contrário, os resultados poderiam não ser corretos. Quando é bem utilizada, esta diretiva garante que só um thread por bloco ordered esté em execução enquanto os outros esperam [4].

3 Exercício 2

Modifique o programa mult.c, que realiza a multiplicação de 2 matrizes. Modifique este código para que ele realize a multiplicação utilizando as primitivas de paralelização de OpenMP. Compare o desempenho com 1, 2, 3 4, 8 e 16 threads. Tente realizar a paralelização no laço for das variáveis i, j e k, explicando no relatório se o comportamento obtido está correto ou não. Apresente e descreva no relatório grafos, tabelas e estatísticas dos tempos de execução.

- **Compare o desempenho obtido, explicando por que melhorou ou piorou e compare também a execução do programa em sua versão sequencial.**
- **Um dos objetivos e verificar se algum overhead e inserido pelo ambiente de execução (runtime OpenMP) quando a versão paralela do programa em OpenMP executa apenas com uma (1) thread, comparando-se com a versão sequencial (mult.c).**
- **Esse programa deve ser executado em pelo menos dois processadores diferentes, para efeitos de encontrar os intervalos de confiança cada execução deve ser repetida pelo menos 10 vezes.**
- **Outras informações que julgarem pertinentes ao contexto do trabalho podem ser adicionadas, e poderão ser somadas como pontos adicionais do EP.**

3.1 Experimentos

Os experimentos realizados neste exercício podem ser separados em duas categorias: a primeira consta da adição de diretivas OpenMP para criar as zonas paralelas, a segunda consta tanto da adição das diretivas como de câmbios no mesmo código para otimizar o uso dos recursos. Cada uma de estas categorias conta com 4 programas diferentes: o programa sequencial, o programa com paralelização no laço da variável i, o programa com paralelização no laço da variável j, e finalmente, o programa com paralelização no laço da variável k. Cada programa, a excepção do programa sequencial, foi executado com diferentes números de threads.

Algumas das características importantes dos experimentos são:

- Otimização de código. Certas partes do programa original mult.c foram cambiadas para melhorar o uso dos recursos. Estas otimizações foram dois:
  - Variável temporal. Uma variável temporal, *tmp*, foi criada para armazenar a suma das multiplicações no laço k.

  - Método de transposição. Um método de transposição foi implementado para obter um acesso sequencial da matriz b no laço k.

- Zonas paralelas. Diferentes zonas foram criadas nos laços das variáveis i, j e k.
- Número de threads. O desempenho foi avaliado com 1, 2, 3, 4, 8 e 16 threads.
- Tipo de programação do for. Foi empleado o tipo *static* ppois o conteúdo de cada um dos loops é semelhante um do outro e por tanto o overhead para determinar qual dos thread vai executar a seguinte carga de trabalho não parece ser necessaria.
- Processadores. Foram utilizados o: **Intel Core i5-3317U CPU @ 1.70GHz x 4** e o **Intel Xeon CPU E5-2630L v2 @ 2.40GHz**.

O código de cada um dos experimentos pode ser observado no Anexo 1 e 2. Sendo o Anexo 1 o conjunto de códigos com adições de diretivas de paralelização; e o Anexo 2 os códigos com diretivas de paralelização, além das otimizações mencionadas previamente.

3.2 Resultados

Os resultados nos tempos de execução dos experimentos da primeira categoria, i.e. os programas com diretivas OpenMP sem otimizações, podem ser enxergados nas tabelas **??** e **??**:

# Threads	Sequencial	Paralela i	Paralela j	Paralela k
1	1.4042	0.7065	0.7559	0.7025
2	1.4042	0.7169	0.7603	0.6920
3	1.4042	0.7171	0.7662	0.7268
4	1.4042	0.7137	0.7616	0.6976
8	1.4042	0.7074	0.7573	0.7060
16	1.4042	0.7129	0.7566	0.6960

(a) Intel Core i5

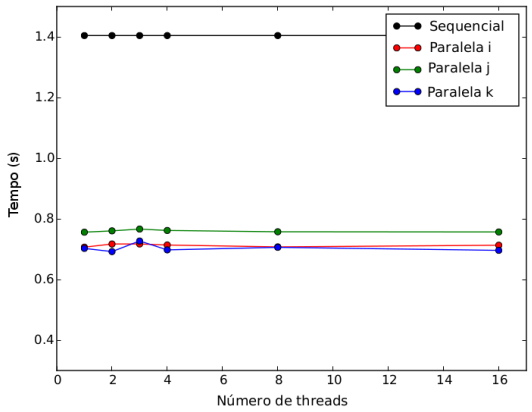
# Threads	Sequencial	Paralela i	Paralela j	Paralela k
1	2.3469	1.2561	1.2747	0.6990
2	2.3469	1.1427	1.1572	0.6424
3	2.3469	1.1389	1.1804	0.6402
4	2.3469	1.2746	1.1563	0.8217
8	2.3469	1.1286	1.1179	0.7177
16	2.3469	1.1921	1.1979	0.6943

(b) Intel Xeon

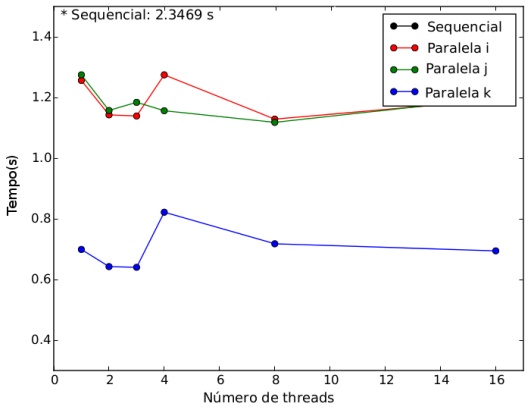
Tabela 1: Diretivas OpenMP, sem Otimizações

Os graficos dos tempos de execução previamente presentados podem ser observado nas figuras 1a e 1b.

Em relação à quantidade de threads, é visível que no caso da Core i5 os tempos se mantém semelhantes; porém, no caso da Xeon os melhores tempos são obtidos quando o número de threads é 2-3. Também pode-se perceber que o tempo aumentou de maneira significativa quando o número de threads foi 4. O que podemos concluir de estes resultados é que no casó da Core i5, que possui 4 cores, os tempos não mudam muito pois o hardware está preparado para a paralelização. No caso da Xeon, que



(a) Intel Core i5



(b) Intel Xeon

Figura 1: Diretivas OpenMP, sem otimizações

também possui 4 cores, os tempos mudam mais significativamente, provavelmente debido ao overhead por cambio de contexto. O overhead pelo uso das diretivas não é signficativo neste caso pois ainda com um (1) thread o tempo é menor em ambos processadores.

Em relação aos laços onde a paralelização foi efetuada, pode ser enxergado que a paralelização no laço da variável k apresenta um tempo menor de execução. Isto pode ser devido ao uso de uma variável temporal *temp* necessaria para poder realizar a paralelização. Ao adicionar esta variável se libera a restrição de ter que acceder à matriz c cada vez que se quer adicionar uma multiplicação. Além disso, em ambos processadores, os tempos de execução são menores quando a paralelização é ubicada no laço da variável i, uma unica excepção é no caso da Xeon com 4 threads. Isto é devido a que quando o laço i é paralelizado a carga de trabalho é significativa em comparação aos outros laços.

Os resultados nos tempos de execução dos experimentos da segunda categoria, foram os seguintes:

# Threads	Sequencial	Paralela i	Paralela j	Paralela k
1	0.5229	0.3506	0.4999	0.6410
2	0.5229	0.3479	0.4848	0.6375
3	0.5229	0.3519	0.4873	0.6349
4	0.5229	0.3500	0.3703	0.6383
8	0.5229	0.3486	0.3468	0.6361
16	0.5229	0.3957	0.3477	0.6318

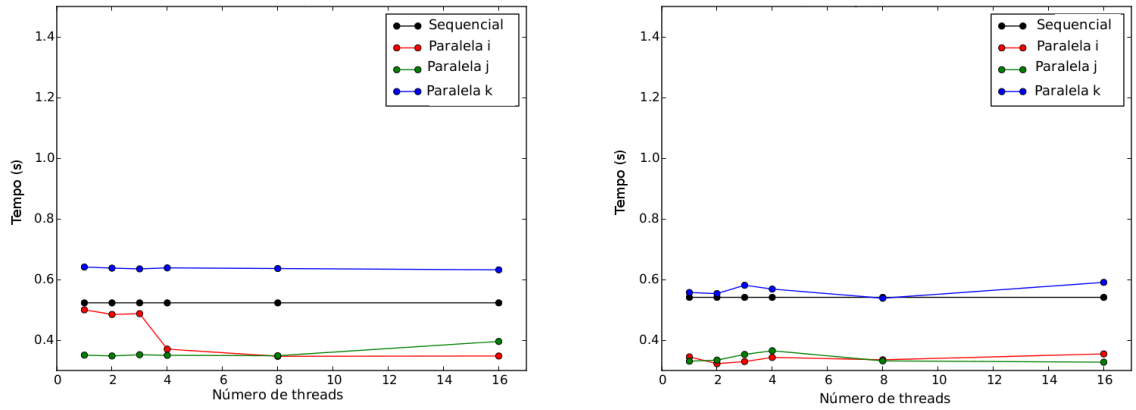
(a) Intel Core i5

# Threads	Sequencial	Paralela i	Paralela j	Paralela k
1	0.5411	0.3448	0.3306	0.5571
2	0.5411	0.3225	0.3341	0.5535
3	0.5411	0.3294	0.3528	0.5808
4	0.5411	0.3431	0.3650	0.5684
8	0.5411	0.3351	0.3316	0.5383
16	0.5411	0.3546	0.3274	0.5904

(b) Intel Xeon

Tabela 2: Diretivas OpenMP, com otimizações

Os grafos dos tempos de execução podem ser observado nas figuras 2a e 2b.



(a) Intel Core i5

(b) Intel Xeon

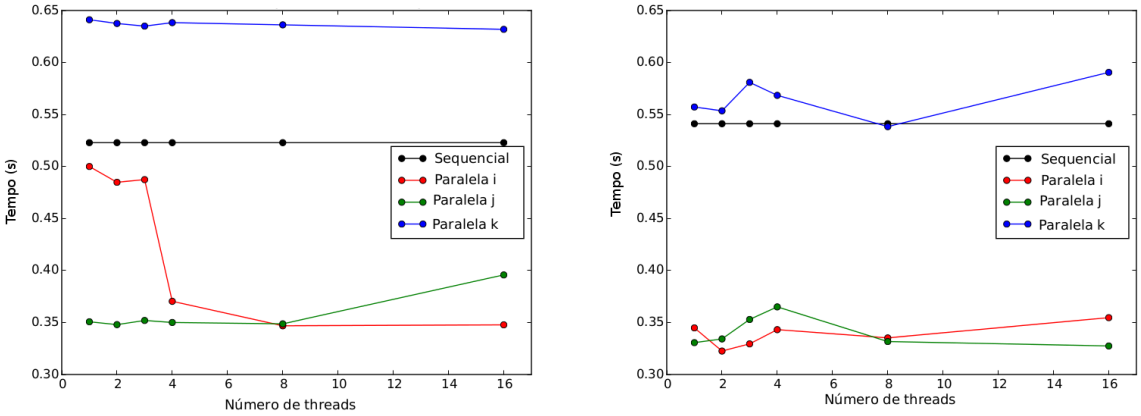
Figura 2: Diretivas OpenMP, com otimizações

Uma das primeras conclusões que pode-se gerar ao observar os tempos dos experimentos com uso de diretivas OpenMP e otimizações de código é: quando se tem em consideração como funciona a memória e como esta é accesada na hora de gerar um programa, o tempo de execução pode diminuir de maneira considerável.

Em ambos processadores é percivivel que o tempo é menor em comparação a suas contra partes sem otimizações de código. Para avaliar melhor os gráficos dos tempos obtidos neste caso se geraram as figuras 3a e 3b, onde o rango de tempo é menor.

Em relação à quantidade de threads, é visível que no caso da Core i5 os tempos se mantém semelhantes no caso da paralelização do laço da varável k; porém, no laço i pode-se enxergar uma grande diminução no tempo quando 4 threads são usados, com 8 e 16 o tempo diminui um pouco mas. No caso da Xeon os melhores tempos são obtidos quando o número de threads é 2 ou 8. O que podemos concluir de estes resultados é que a quantidade de thread vira a ser um aspecto importante e que o melhor valor é 8, isto e posivelmente devido a que os processadores contam com 4 cores, mas no caso da Core i5 também conta com tecnologia hyper-threading. O overhead pelo uso das diretivas só é signficativo em comparação à paralelização no laço k.

Em relação aos laços onde a paralelização foi efetuada, pode ser enxergado que a paralelização no laço da variável k apresenta um tempo maior de execução. Isto é devido ao uso de uma variável temporal tmp em todas as paralelizações. Agora que todas as paralelizações contam com esta misma otmização pode se observar que o overhead gerado por o cambio de contexto é maior quando a paralelização é gerado no laço mais interno. Além disso, em ambos processadores, os tempos de execução são menores quando a paralelização é ubicada no laço da variável i ou j. Sendo os menores tempos obtidos quando a quantidade de threads é 8 e os tempos não variam



(a) Intel Core i5

(b) Intel Xeon

Figura 3: Diretivas OpenMP, com otimizações (Zoom)

muito entre a paralelização no laço i e o laço j.

3.3 Conclusões

O que pode-se concluir dos experimentos é que a paralelização ajuda na melhoria dos tempos de execução. Porém, os cambios no código para facilitar a paralelização e o acceso a memoria podem gerar melhores resultados. O tempo de execução da version sequencial (sem diretivas OpenMP) foi menor do que os tempos das versões paralelizadas que não contiam otimizações.

Além disso, o overhead por o cambio de contexto e maior quando a paralelização é criada nos laços internos, onde a carga de trabalho é muito menor. Ao ter pouco trabalho os threads executam este de maneira rapida mas debem trocar os valores dos registros, etc. para dar espaço a outro thread. Este cambio de contexto ocorre mais constantemente quando a paralização é levada a cabo no laço interno.

O número de threads não brindo muita informação, salvo no caso do uso de 8 threads. Em este caso, os tempos foram melhores no Core i5 pois em teoria a sistema operativo assume que conta com 8 cores para trabalhar.

4 Conclusões

Nossas conclusões finais são:

- É muito importante que se tenha em consideração o modo em que a memoria é accesada para garantir o melhor tempo de execução.
- Para evitar o overhead, a carga de cada um dos threads debe ser consideravel. Quando em dúvida, o laço exterior é a melhor opção.
- Para a escolha do número de threads é importante conhecer a arquitetura no qual será executado o programa. Quando esta informação não é conhecida, a melhor opção é assumir que o tempo não será o ótimo.
- Na uso das diretivas OpenMP é importante AQUI LLENATUCONSLUSION!!!!!!!!!!!!!!!!!!!!

Referências

[1] OpenMP ARB Corporation. Frequently asked questions openmp, 2013.

[2] F. Gebali. *Algorithms and parallel computing.* Wiley, 2011.

[3] A. Gottlieb and G. Almasi. *Highly parallel computing.* Benjamin-Cummings Publishing Co., 1989.

[4] Lawrence Livermore National Lab. Introduction to openmp, 2014.

[5] N. Matloff. *Programming on parallel machines.* University of California, Davis, 2014.

[6] Alfredo Goldman Rogério A. Gonçalves. Introdução ao openmp, 2015.

A Anexo I

Código do programa mult.c sequencial com trocas para a medição do tempo.

```
/*
ID: diana.n1
PROG: MultSeq
LANG: C++
*/

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#include "omp.h"


int main(int argc, char **argv) {
    double start, end;
    start = omp_get_wtime();
    long **a, **b, **c;
    int N = 500;

    if (argc == 2) {
        N = atoi (argv[1]);
        assert (N > 0);
    }
```

```
    int i,j,k,mul=5;
    long col_sum = N * (N-1) / 2;


    a = (long **)malloc (N * sizeof(long *));
    b = (long **)malloc (N * sizeof(long *));
    c = (long **)malloc (N * sizeof(long *));

    for (i=0; i<N; i++) {
        a[i] = (long *)malloc (N * sizeof(long));
        b[i] = (long *)malloc (N * sizeof(long));
        c[i] = (long *)malloc (N * sizeof(long));
    }


    for (i=0; i<N; i++)
        for (j=0; j<N; j++) {
        a[i][j] = i*mul;
         b[i][j] = i;
        c[i][j] = 0;
        }

    printf ("Matrix generation finished.\n");

    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
        for (k=0; k<N; k++)
         c[i][j] += a[i][k] * b[k][j];

    printf ("Multiplication finished.\n");

    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
        assert ( c[i][j] == i*mul * col_sum);
        printf ("Test finished.\n");

    end = omp_get_wtime();
    printf("Time: %lf.\n", end-start);
}
```

Código do programa mult.c com paralelização no laço da variável i.

```
/*
ID: diana.n1
PROG: MultParalleli
LANG: C++
*/

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#include "omp.h"

int main(int argc, char **argv) {
    double start, end;
    start = omp_get_wtime();
    int nthreads, tid, chunk = 10;

    long **a, **b, **c;
    int N = 500;

    if (argc == 2) {
        N = atoi (argv[1]);
        assert (N > 0);
    }

    int i,j,k,mul=5;
    long col_sum = N * (N-1) / 2;

    a = (long **)malloc (N * sizeof(long *));
    b = (long **)malloc (N * sizeof(long *));
    c = (long **)malloc (N * sizeof(long *));

    for (i=0; i<N; i++) {
        a[i] = (long *)malloc (N * sizeof(long));
        b[i] = (long *)malloc (N * sizeof(long));
        c[i] = (long *)malloc (N * sizeof(long));
    }

    #pragma omp parallel shared (a,b,c,nthreads, chunk) private (i, j,k,tid)
    {
```

```
        tid = omp_get_thread_num();
        #pragma omp single
            printf("Number threads = %d\n", omp_get_num_threads());

        #pragma omp for schedule(static, chunk)
            for (i=0; i<N; i++){
                for (j=0; j<N; j++) {
                    a[i][j] = i*mul;
                    b[i][j] = i;
                    c[i][j] = 0;
                }
            }

        #pragma omp single
            printf ("Matrix generation finished.\n");

        #pragma omp for schedule(static, chunk)
            for (i=0; i<N; i++){
                for (j=0; j<N; j++){
                    for (k=0; k<N; k++){
                        c[i][j] += a[i][k] * b[k][j];
                    }
                }
            }

        #pragma omp single
            printf ("Multiplication finished.\n");

        #pragma omp for schedule(static, chunk)
            for (i=0; i<N; i++){
                for (j=0; j<N; j++){
                    assert ( c[i][j] == i*mul * col_sum);
                }
            }
        #pragma omp single
            printf ("Test finished.\n");
    }
    end = omp_get_wtime();
    printf("Time: %lf.\n", end-start);
}
```

Código do programa mult.c com paralelização no laço da variável j.

```
/*
ID: diana.n1
PROG: MultParallelj
LANG: C++
*/

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#include "omp.h"

int main(int argc, char **argv) {
    double start, end;
    start = omp_get_wtime();
    int nthreads, tid, chunk = 10;

    long **a, **b, **c;
    int N = 500;

    if (argc == 2) {
        N = atoi (argv[1]);
        assert (N > 0);
    }

    int i,j,k,mul=5;
    long col_sum = N * (N-1) / 2;

    a = (long **)malloc (N * sizeof(long *));
    b = (long **)malloc (N * sizeof(long *));
    c = (long **)malloc (N * sizeof(long *));

    for (i=0; i<N; i++) {
        a[i] = (long *)malloc (N * sizeof(long));
        b[i] = (long *)malloc (N * sizeof(long));
        c[i] = (long *)malloc (N * sizeof(long));
    }

    #pragma omp parallel shared (a,b,c,nthreads, chunk) private (i, j,k,tid)
```

```
    {
        tid = omp_get_thread_num();
        #pragma omp single
            printf("Number threads = %d\n", omp_get_num_threads());

        #pragma omp for schedule(static, chunk)
            for (i=0; i<N; i++){
                for (j=0; j<N; j++) {
                    a[i][j] = i*mul;
                    b[i][j] = i;
                    c[i][j] = 0;
                   }
            }
        #pragma omp single
             printf ("Matrix generation finished.\n");

        for (i=0; i<N; i++){
             #pragma omp for schedule(static, chunk)
                 for (j=0; j<N; j++){
                     for (k=0; k<N; k++){
                         c[i][j] += a[i][k] * b[k][j];
                     }
                 }
         }

        #pragma omp single
            printf ("Multiplication finished.\n");

        #pragma omp for schedule(static, chunk)
            for (i=0; i<N; i++){
               for (j=0; j<N; j++){
                    assert ( c[i][j] == i*mul * col_sum);
                }
        }
        #pragma omp single
            printf ("Test finished.\n");
    }

    end = omp_get_wtime();
    printf("Time: %lf.\n", end-start);
}
```

Código do programa mult.c com paralelização no laço da variável k.

```
/*
ID: diana.n1
PROG: MultParallelk
LANG: C++
*/

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#include "omp.h"

int main(int argc, char **argv) {
    double start, end;
    start = omp_get_wtime();
    int nthreads, tid, chunk = 10;

    long **a, **b, **c;
    int N = 500;

    if (argc == 2) {
        N = atoi (argv[1]);
        assert (N > 0);
    }

    int i,j,k,mul=5;
    long col_sum = N * (N-1) / 2;
    long temp;

    a = (long **)malloc (N * sizeof(long *));
    b = (long **)malloc (N * sizeof(long *));
    c = (long **)malloc (N * sizeof(long *));

    for (i=0; i<N; i++) {
        a[i] = (long *)malloc (N * sizeof(long));
        b[i] = (long *)malloc (N * sizeof(long));
        c[i] = (long *)malloc (N * sizeof(long));
```

```
}

#pragma omp parallel shared (a,b,c,nthreads, chunk, temp) private (i, j,k,tid)
{
    tid = omp_get_thread_num();
    #pragma omp single
        printf("Number threads = %d\n", omp_get_num_threads());

    #pragma omp for schedule(static, chunk)
    for (i=0; i<N; i++){
        for (j=0; j<N; j++) {
            a[i][j] = i*mul;
            b[i][j] = i;
            c[i][j] = 0;
        }
    }

    #pragma omp single
        printf ("Matrix generation finished.\n");

    for (i=0; i<N; i++){
        for (j=0; j<N; j++){
            #pragma omp single
                temp = 0;
            #pragma omp for schedule(static, chunk) reduction (+:temp)
                for (k=0; k<N; k++){
                    temp += a[i][k] * b[k][j];
                }
            #pragma omp single
                c[i][j] = temp;
        }
    }

    #pragma omp single
        printf ("Multiplication finished.\n");

    #pragma omp for schedule(static, chunk)
        for (i=0; i<N; i++){
            for (j=0; j<N; j++){
                assert ( c[i][j] == i*mul * col_sum);
            }
```

```
            }
        #pragma omp single
            printf ("Test finished.\n");
    }
    end = omp_get_wtime();
    printf("Time: %lf.\n", end-start);
}
```

B Anexo II

Código do programa mult.c sequencial com optimizações de código.

```
/*
ID: diana.n1
PROG: MultSeqCodeChange
LANG: C++
*/

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#include "omp.h"

void transpose(int n, long ** m){
    long tmp;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            tmp = m[i][j];
            m[i][j] = m[j][i];
            m[j][i] = tmp;
        }
    }
}

int main(int argc, char **argv) {
    double start, end;
    start = omp_get_wtime();
    long **a, **b, **c;
```

```
int N = 500;

if (argc == 2) {
    N = atoi (argv[1]);
    assert (N > 0);
}

int i,j,k,mul=5;
long col_sum = N * (N-1) / 2, tmp;


a = (long **)malloc (N * sizeof(long *));
b = (long **)malloc (N * sizeof(long *));
c = (long **)malloc (N * sizeof(long *));
for (i=0; i<N; i++) {
    a[i] = (long *)malloc (N * sizeof(long));
    b[i] = (long *)malloc (N * sizeof(long));
    c[i] = (long *)malloc (N * sizeof(long));
}


for (i=0; i<N; i++){
    for (j=0; j<N; j++) {
        a[i][j] = i*mul;
        b[i][j] = i;
        c[i][j] = 0;
    }
}

printf ("Matrix generation finished.\n");

transpose(N, b);
for (i=0; i<N; i++){
   for (j=0; j<N; j++){
        tmp = 0;
        for (k=0; k<N; k++){
            tmp += a[i][k] * b[j][k];
        }
        c[i][j] = tmp;
   }
}
```

```
    printf ("Multiplication finished.\n");

    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            assert ( c[i][j] == i*mul * col_sum);

    printf ("Test finished.\n");

    end = omp_get_wtime();
    printf("Time: %lf.\n", end-start);
}
```

Código do programa mult.c com paralelização no laço da variável i com optimizações de código.

```
/*
ID: diana.n1
PROG: MultParalleliCodeChange
LANG: C++
*/

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#include "omp.h"

void transpose(int n, long ** m){
    long tmp;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            tmp = m[i][j];
            m[i][j] = m[j][i];
            m[j][i] = tmp;
        }
    }
}
```

```
int main(int argc, char **argv) {
    double start, end;
    start = omp_get_wtime();
    int nthreads, tid, chunk = 10;

    long **a, **b, **c;
    int N = 500;

    if (argc == 2) {
        N = atoi (argv[1]);
        assert (N > 0);
    }

    int i,j,k,mul=5;
    long col_sum = N * (N-1) / 2, tmp;

    a = (long **)malloc (N * sizeof(long *));
    b = (long **)malloc (N * sizeof(long *));
    c = (long **)malloc (N * sizeof(long *));

    for (i=0; i<N; i++) {
        a[i] = (long *)malloc (N * sizeof(long));
        b[i] = (long *)malloc (N * sizeof(long));
        c[i] = (long *)malloc (N * sizeof(long));
    }

    #pragma omp parallel shared (a,b,c,nthreads, chunk) private (i, j,k,tid, tmp)
    {
        tid = omp_get_thread_num();

        #pragma omp single
            printf("Number threads = %d\n", omp_get_num_threads());

        #pragma omp for schedule(static, chunk)
            for (i=0; i<N; i++){
                for (j=0; j<N; j++) {
                    a[i][j] = i*mul;
                    b[i][j] = i;
                    c[i][j] = 0;
                }
            }
```

```
        #pragma omp single
            printf ("Matrix generation finished.\n");

        #pragma omp single
            transpose(N, b);

        #pragma omp for schedule(static, chunk)
            for (i=0; i<N; i++){
                for (j=0; j<N; j++){
                    tmp = 0;
                    for (k=0; k<N; k++){
                        tmp += a[i][k] * b[j][k];
                    }
                    c[i][j] = tmp;
                }
            }

        #pragma omp single
            printf ("Multiplication finished.\n");

        #pragma omp for schedule(static, chunk)
            for (i=0; i<N; i++){
               for (j=0; j<N; j++){
                    assert ( c[i][j] == i*mul * col_sum);
                }
            }
        #pragma omp single
            printf ("Test finished.\n");
    }
    end = omp_get_wtime();
    printf("Time: %lf.\n", end-start);
}
```

Código do programa mult.c com paralelização no laço da variável j com optimizações de código.

```
/*
ID: diana.n1
PROG: MultParalleljCodeChange
LANG: C++
```

```
*/

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#include "omp.h"

void transpose(int n, long ** m){
    long tmp;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            tmp = m[i][j];
            m[i][j] = m[j][i];
            m[j][i] = tmp;
        }
    }
}

int main(int argc, char **argv) {
    double start, end;
    start = omp_get_wtime();
    int nthreads, tid, chunk = 10;

    long **a, **b, **c;
    int N = 500;

    if (argc == 2) {
        N = atoi (argv[1]);
        assert (N > 0);
    }

    int i,j,k,mul=5;
    long col_sum = N * (N-1) / 2, tmp;

    a = (long **)malloc (N * sizeof(long *));
    b = (long **)malloc (N * sizeof(long *));
    c = (long **)malloc (N * sizeof(long *));

    for (i=0; i<N; i++) {
```

```
    a[i] = (long *)malloc (N * sizeof(long));
    b[i] = (long *)malloc (N * sizeof(long));
    c[i] = (long *)malloc (N * sizeof(long));
}

#pragma omp parallel shared (a,b,c,nthreads, chunk) private (i,  j, k, tid, tmp)
{
    tid = omp_get_thread_num();
    #pragma omp single
        printf("Number threads = %d\n", omp_get_num_threads());

    #pragma omp for schedule(static, chunk)
        for (i=0; i<N; i++){
            for (j=0; j<N; j++) {
                a[i][j] = i*mul;
                b[i][j] = i;
                c[i][j] = 0;
            }
        }
    #pragma omp single
        printf ("Matrix generation finished.\n");

    #pragma omp single
        transpose(N, b);

    for (i=0; i<N; i++){
        #pragma omp for schedule(static, chunk)
            for (j=0; j<N; j++){
                tmp = 0;
                for (k=0; k<N; k++){
                    tmp += a[i][k] * b[j][k];
                }
                c[i][j] = tmp;
            }
    }

    #pragma omp single
        printf ("Multiplication finished.\n");

    #pragma omp for schedule(static, chunk)
        for (i=0; i<N; i++){
```

```
                for (j=0; j<N; j++){
                     assert ( c[i][j] == i*mul * col_sum);
                 }
            }
        #pragma omp single
            printf ("Test finished.\n");
    }
    end = omp_get_wtime();
    printf("Time: %lf.\n", end-start);
}
```

Código do programa mult.c com paralelização no laço da variável k com optimizações de código.

```
/*
ID: diana.n1
PROG: MultParallelkCodeChange
LANG: C++
*/

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#include "omp.h"

void transpose(int n, long ** m){
    long tmp;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j < n; j++){
            tmp = m[i][j];
            m[i][j] = m[j][i];
            m[j][i] = tmp;
        }
    }
}

int main(int argc, char **argv) {
    double start, end;
    start = omp_get_wtime();
```

```
int nthreads, tid, chunk = 10;

long **a, **b, **c;
int N = 500;

if (argc == 2) {
    N = atoi (argv[1]);
    assert (N > 0);
}

int i,j,k,mul=5;
long col_sum = N * (N-1) / 2, tmp;

a = (long **)malloc (N * sizeof(long *));
b = (long **)malloc (N * sizeof(long *));
c = (long **)malloc (N * sizeof(long *));

for (i=0; i<N; i++) {
    a[i] = (long *)malloc (N * sizeof(long));
    b[i] = (long *)malloc (N * sizeof(long));
    c[i] = (long *)malloc (N * sizeof(long));
}

#pragma omp parallel shared (a,b,c,nthreads, chunk, tmp) private (i, j,k,tid)
{
    tid = omp_get_thread_num();
    #pragma omp single
        printf("Number threads = %d\n", omp_get_num_threads());

    #pragma omp for schedule(static, chunk)
        for (i=0; i<N; i++){
            for (j=0; j<N; j++) {
                a[i][j] = i*mul;
                b[i][j] = i;
                c[i][j] = 0;
       }
        }
    #pragma omp single
        printf ("Matrix generation finished.\n");

    #pragma omp single
```

```
            transpose(N, b);

        for (i=0; i<N; i++){
            for (j=0; j<N; j++){
                #pragma omp single
                    tmp = 0;
                #pragma omp for schedule(static, chunk) reduction (+:tmp)
                    for (k=0; k<N; k++){
                        tmp += a[i][k] * b[j][k];
                    }
                #pragma omp single
                    c[i][j] = tmp;
           }
        }

        #pragma omp single
            printf ("Multiplication finished.\n");

        #pragma omp for schedule(static, chunk)
            for (i=0; i<N; i++){
               for (j=0; j<N; j++){
                    assert ( c[i][j] == i*mul * col_sum);
                }
            }
        #pragma omp single
            printf ("Test finished.\n");
    }
    end = omp_get_wtime();
    printf("Time: %lf.\n", end-start);
}
```