Universidade do Minho

Mestrado em Engenharia Informática

Algoritmos Paralelos - 2014/2015

Portfolio

18 de Junho de 2015

Fábio Gomes pg27752

Índice

BubbleSort Paralelizado - Introdução

O problema que nos foi apresentado está relacionado com o algoritmo de ordenação `BubbleSort` normalmente conhecido como o pior algoritmos de todos do género. É nosso trabalho tentar paraleliza-lo.

Há que ter em atenção pois não se trata de uma simples paralelização com `pragma omp parallel` tradicional, temos que tratar de casos de sincronismo e concorrência de dados.

Caracterização do Sistema

Nodo do Cluster obtido

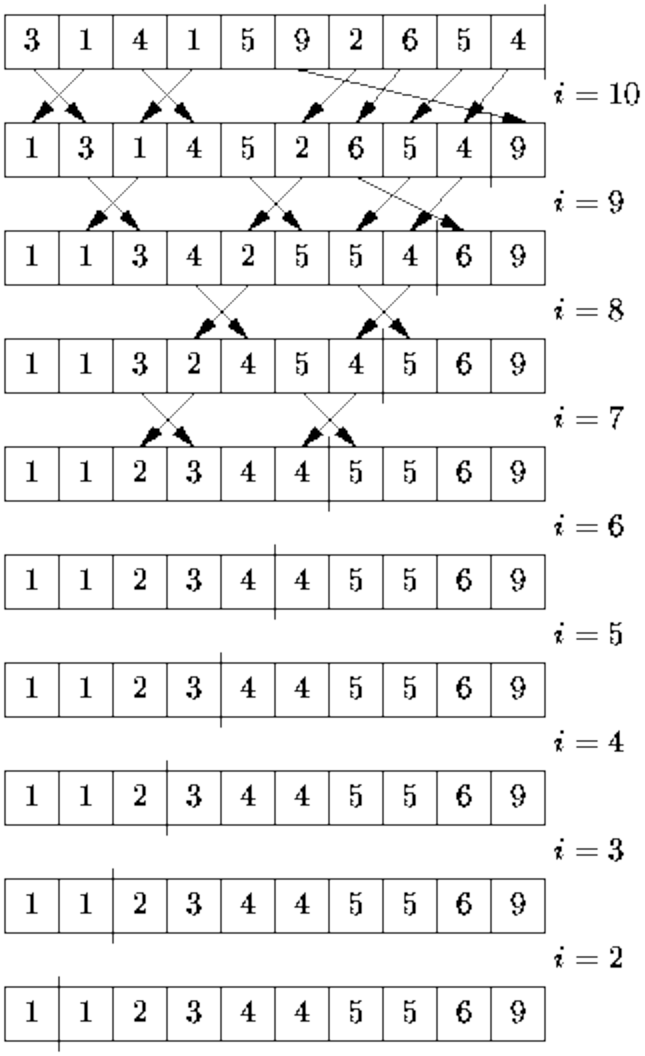
Para utilização deste projeto tivemos que recorrer ao SEARCH, que tem a seguinte especificação.

	Cluster Node
Manufacturer	intel
Model	e5-2695v2
Clock speed	2.4 GHz
Architecture	x86-64
µArchitecture	Ivy Bridge
#Cores	12
#Threads per Core	2
Total Threads	24
Peak FP performance	38.4 GFLOP/s
L1 cache	768kB (32kB L1 Data per Core)
L2 cache	3MB (256kB per Core)
L3 cache	30,720 kB (Shared)
Main Memory	66,068,588 kB
Memory Channels	4
Max Memory	786,432 MB
Max Bandwidth	59,733.32 MB/s

Explicação do Problema

O BubbleSort é uma técnica básica em que se comparam dois elementos do vetor/*array* e trocam-se as suas posições se o primeiro elemento é maior do que o segundo. São feitas várias passagens pela tabela e em cada uma, comparam-se dois elementos adjacentes tal que se estes elementos estiverem fora de ordem, eles são trocados segundo o critério previamente explicado. Podendo ser trocado conforme se pretenda de forma Ascendente ou Descendente.

Este método tem como vantagem a sua simplicidade e a estabilidade mas peca por ser muito lento. Deve ser usado para tabelas muito pequenas ou quando se sabe que a tabela está quase ordenada. O nome vem do método da troca em que os elementos menores (mais "leves") vão aos poucos "subindo" para o início da tabela, como se fossem bolhas. Exemplificado na figura seguinte.



Exemplo para um array de 10 elementos - 1

Análise do Código Fornecido

O programa é bastante simples, é gerado um vetor `a` de tamanho `n` e para cada posição dele é percorrido até ao fim fazendo as trocas. Terminando apenas quando todos os índices estiverem percorridos.

```
/*-------------------------------------------------------------------
 * Function:     Bubble_sort
 * Purpose:      Sort list using bubble sort
 * In args:      n
 * In/out args:  a
 */
void Bubble_sort(
      int  a[]  /* in/out */,
      int  n    /* in     */) {
   int list_length, i, temp;

   for (list_length = n; list_length >= 2; list_length--)
      for (i = 0; i < list_length-1; i++)
         if (a[i] > a[i+1]) {
            temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
         }

}  /* Bubble_sort */
```

Paralelização com OpenMP

A Paralelização não é tão trivial como poderá parecer porque há concorrência nos dados. Para tal é necessário usar um novo mecanismo do OpenMP chamado `omp_lock_t`. Como o nome indica é feito um lock a um elemento de forma a que apenas 1 thread o consiga ter em sua posse. Assim, crio um array de locks chamado lock de tamanho igual ao número de threads.

```
lock = (omp_lock_t*) malloc((nthreads)*sizeof(omp_lock_t));
```

Com ele já teremos mais controlo no vetor para ordenarmos sem problemas.

É preciso ainda definir uma subdivisão do vetor, chamada de bloco. O tamanho de cada bloco `block_size` é igual ao tamanho do vetor a dividir pelo número de locks (igual ao número de threads). Cada bloco é associado um `lock`, assim apenas 1 thread poderá fazer as trocas à vontade sem se preocupar com concorrência pois é apenas ela que lá está.

Com o decorrer do tempo o lock será libertado e o seguinte será adquirido, o que foi libertado será apanhado por uma thread que estava à espera para continuar a sua iteração. Desta forma haverá uma sequência de threads ao longo do vetor sem nunca se cruzarem causando conflitos.

Em termos de código, é iniciado o vetor de locks, algumas variáveis de condição e o tamanho do bloco.

```
double start = omp_get_wtime();
    int list_length=n, i,x,ind, temp,ltemp,max;
    block_size = (int)(list_length/nthreads);
    for(i=0;i<nthreads;i++)
    omp_init_lock(&(lock[i]));
    int ordered = 0,changed = 0;
```

Depois começa o ciclo principal com um pragma omp parallel para criar a região paralela com a particularidade de partilhar, obviamente, o vetor a ser ordenado, o tamanho da lista que vai decrescendo ao longo das iterações, o vetor de locks e a condição ordered que nos vai parar o ciclo principal quando 1 thread chegar ao fim sem fazer uma troca sugerindo que está o vetor ordenado poupando tempo.

```
#pragma omp parallel shared(a,lock,n,list_length,ordered) num_threads(nthreads)
private(temp,i,ltemp,ind) firstprivate(block_size,changed)
    while(!ordered){
changed = 0;
```

Já dentro do ciclo while temos que percorrer os blocos do vetor e em cada um fazer lock. Depois apenas iteramos elementos naquelas posições até ao fim do bloco, ou do vetor quando estamos no último bloco. Por fim libertamos o lock e verificamos se fizemos alterações para parar o ciclo while e decrementamos 1 ao tamanho do vetor para que a próxima thread não vá até ao fim de todo pois é inútil.

```
      for(ind=0;ind<nthreads;ind++){
            omp_set_lock(&(lock[ind]));
            max = list_length>(ind+1)*block_size ? (ind+1)*block_size :
list_length;
            for (i = ind*block_size; i < max; i++){
                if (a[i] > a[i+1]) {
                    temp = a[i];
                    a[i] = a[i+1];
                    a[i+1] = temp;
                    changed = 1;
                }
            }
            omp_unset_lock(&(lock[ind]));
        }
        if(changed==0)
        ordered = 1;
        list_length--;
    }
```

Para finalizar destruímos o vetor de locks e contabilizamos o tempo.

```
    for(i=0;i<nthreads;i++)
    omp_destroy_lock(&(lock[i]));
    double end = omp_get_wtime();
    printf("%3.2gn",end-start);
```

Código BubbleSort Paralelizado

```
void Bubble_sort(
int  a[]  /* in/out */,
int  n    /* in     */) {
    double start = omp_get_wtime();
    int list_length=n, i,x,ind, temp,ltemp,max;
    block_size = (int)(list_length/nthreads);
    for(i=0;i<nthreads;i++)
    omp_init_lock(&(lock[i]));
    int ordered = 0,changed = 0;
    #pragma omp parallel shared(a,lock,n,list_length,ordered) num_threads(nthreads)
private(temp,i,ltemp,ind) firstprivate(block_size,changed)
    while(!ordered){
        changed = 0;
        for(ind=0;ind<nthreads;ind++){
            omp_set_lock(&(lock[ind]));
            max = list_length>(ind+1)*block_size ? (ind+1)*block_size : list_length;
            for (i = ind*block_size; i < max; i++){
                if (a[i] > a[i+1]) {
                    temp = a[i];
                    a[i] = a[i+1];
                    a[i+1] = temp;
                    changed = 1;
                }
            }
            omp_unset_lock(&(lock[ind]));
        }
        if(changed==0)
        ordered = 1;
        list_length--;
    }
    for(i=0;i<nthreads;i++)
    omp_destroy_lock(&(lock[i]));
    double end = omp_get_wtime();
    printf("%3.2gn",end-start);
}  /* Bubble_sort */
```

Testes e Análise de Resultados

Com tudo concluído, é tempo de fazer medições de tempos de execução e analisar os resultados.

Os testes foram feitos para uma divisão de blocos igual ao número de threads apenas e com número de threads e de tamanho variado como sugere a seguinte tabela.

	#Threads											
Size	**Serial**	**2**	**4**	**6**	**8**	**10**	**14**	**16**	**20**	**30**	**40**	Min
100	**0,002**	0,0002	0,0003	0,0003	0,0004	0,0007	0,0006	0,0006	0,0008	0,0012	0,0065	0,0002
500	**0,004**	0,0008	0,0018	0,0018	0,0019	0,003	0,0022	0,002	0,0021	0,0026	0,0079	0,0008
1000	**0,008**	0,0035	0,0043	0,0047	0,0054	0,0086	0,0047	0,005	0,0048	0,0047	0,016	0,0035
5000	**0,092**	0,069	0,071	0,073	0,068	0,067	0,062	0,05	0,044	0,041	0,036	0,036
10000	**0,365**	0,18	0,28	0,27	0,25	0,22	0,18	0,16	0,14	0,11	0,13	0,11
50000	**8,905**	7,5	6,7	6,6	6,5	6,4	3,2	3	2,4	2,3	1,9	1,9
100000	**30,135**	17	27	26	24	14	13	11	9,9	7,9	7,5	7,5
200000	**122,07**	90	110	120	90	67	56	44	42	31	21	21

A Verde estão representados os tempos mais baixos (melhores) e a Vermelho os maiores (piores).

Conseguimos traçar um padrão, a versão Serial fornecida consegue ser sempre pior para estes casos de teste mas prevejo que para valores mais baixos de input isto não ocorra.

Como seria de esperar quanto maior o problema em termos de tamanho do vetor mais tempo irá durar, obrigando a um aumento do número de threads para que o seu tempo de execução não aumente também de forma explosiva.

Todos os tempos medidos são abaixo do Serial o que me satisfaz mas muito provavelmente será possível obter tempos muito melhores porque há algumas lacunas na minha implementação que não foram corrigidas, nomeadamente quando uma thread está à espera de um lock para um bloco seguinte mas esse bloco já não é preciso pois o tamanho da lista (virtual que decresce a cada thread concluída) já é mais baixo que esse. Fazendo com que seja uma perca de tempo esperar pelo lock que não é necessário de todo.

Conclusão

Para além de ser um caso prático e cativar por si só foi também um trabalho muito bom na medida em que permitiu usar novos conhecimentos adquiridos nesta Unidade Curricular que não foram abordados anteriormente. Um trabalho que apesar de parecer simples engloba em si vários aspectos que tiveram de ser devidamente considerados para que o algoritmo funcionasse corretamente.

No final dou o trabalho por concluído mas sem deixar a nota que poderá ser melhorado como já referi. Já tínhamos utilizado locks em Sistemas Distribuídos mas nesta aplicação prática revelaram-se muito mais críticos e obrigaram a uma maior preparação e análise do problema.

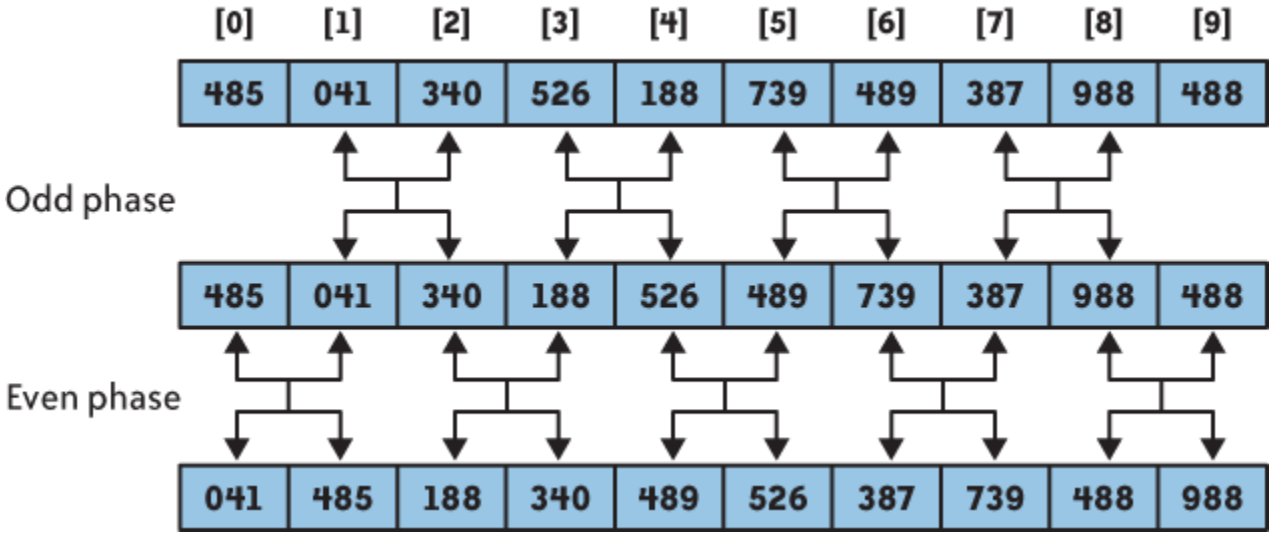
Odd-Even Sort Paralelizado - Introdução

O problema que nos foi apresentado está relacionado com o algoritmo de ordenação `Odd-Even Sort` que se baseia nas posições Ímpares e Pares. É nosso trabalho tentar paraleliza-lo.

Explicação do Problema

O *Odd-Even Sort* foi desenvolvido para tirar partido do paralelismo e das interconexões. O seu funcionamento é reduzido a 2 iterações que passam por comparar os índices Ímpares e os Pares com o elemento seguinte até que não sejam precisas mais trocas ficando ordenado.

O esquema seguinte explica o seu funcionamento.



Exemplo para um array de 10 elementos - 2

Análise do Código Fornecido

Este é o método `OESort`, o principal do algoritmo. Enquanto hajam mudanças o `exch` é 1 e a cada passagem no while é feita a troca do índice de paridade inicial (`start`).

```
void OESort(int NN, int *A)
{
    int exch = 1, start = 0, i;
    int temp;

    while (exch || start) {
        exch = 0;
        for (i = start; i < NN-1; i+=2) {
            if (A[i] > A[i+1]) {
                temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
                exch = 1;
            }
        }
        if (start == 0) start = 1;
        else start = 0;
    }
}
```

Paralelização com OpenMP – Versão 1

A Paralelização passou por criar uma primeira versão (v1) em que o exch agora passa a ter o número de threads e em cada iteração é decrementado 1 por cada thread. Se alguma alteração se realizar o `exch` passa para o número de threads para que na próxima iteração do while a condição de paragem falhe e o programa continue a tentar ordenar. A barreira depois de decrementar o `exch` tem que estar presente pois pode acontecer o caso em que o programa nunca mais termina pois o exch é alterado quando é feita uma alteração e ainda há uma thread atrasada a fazer o `exch--`. O `pragma omp single` no fim faz barreira e troca a paridade, assim no início do ciclo seguinte todas as threads sabem que índice pegar.

```
void OESort(int NN, int *A){
    int exch = 1, start = 0, i;
    #pragma omp parallel
    {int temp;
     exch = omp_get_num_threads();
    while (1) {
        if(exch <= 0 && start == 0) break;
        #pragma omp critical
        exch--;
        #pragma omp barrier
        #pragma omp for
        for (i = start; i < NN-1; i+=2) {
            if (A[i] > A[i+1]) {
                temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
                #pragma omp critical
                exch = omp_get_num_threads();}
        }
        #pragma omp single
        if (start == 0) start = 1;
        else start = 0;
     }}}
```

Paralelização com OpenMP – Versão 2

Esta Paralelização tem como objetivo reduzir o *overhead* causado pelo contínuo *sleep/wake* das threads. Ao fazer unroll aos 2 ciclos for que são indiretamente executados devido à paridade do algoritmo conseguimos fazer mais trabalho aproveitando o tempo em que a thread está ativa e eventualmente melhor performance.

São necessárias variáveis extra, `exch0` referente aos exchanges/trocas do primeiro ciclo for, a `exch1` para o segundo e a `first` que apenas serve para indicar o segundo ciclo que é a primeira iteração de trocas e se não existirem mudanças para não estranhar e tentar. Se fizer alguma alteração no primeiro ciclo `exch0` é posto a 1, baseado nisso o segundo ciclo apenas é executado se de facto houve alterações porque o `exch0` é 1. Se não trocarmos em nenhum dos ciclos o while acaba e damos por terminada a troca.

Esta versão é caracterizada por estar sempre a abrir e a fechar a zona paralela a cada iteração do ciclo while.

```
void OESort_v2(int NN, int *A)
{
    int exch0, exch1=1, i,first=0;
    while (exch1) {
        exch0=0;exch1=0;
        #pragma omp parallel
           {int temp;
            #pragma omp for private(i,temp)
            for (i = 0; i < NN-1; i+=2) {
                if (A[i] > A[i+1]) {
                    temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
                    exch0=1;
                }
            }
            if(exch0 || !first){
            #pragma omp for private(i,temp)
            for (i = 1; i < NN-1; i+=2) {
                if (A[i] > A[i+1]) {
                    temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
                    exch1=1;
                }
            }
               }}
          first=1;
    }}
```

Paralelização com OpenMP – Versão 3

Como a versão anterior pode trazer problemas devido à tal abertura e fecho da zona paralela, fiz uma alteração em que só se abre uma zona, fora do ciclo while, mas leva com 2 barreiras para sincronização. Esta mudança poderá trazer vantagens se os tempos de espera devido à barreira não forem tão altos em comparação à versão anterior.

```
void OESort_v3(int NN, int *A)
{
    int exch0, exch1=1, i,first=0;
    #pragma omp parallel
    {int temp;
    while (exch1) {
        #pragma omp barrier
        exch0=0;exch1=0;
        #pragma omp barrier

        #pragma omp for private(i,temp)
        for (i = 0; i < NN-1; i+=2) {
            if (A[i] > A[i+1]) {
                temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
                exch0=1;
            }
        }
        if(exch0 || !first){
        #pragma omp for private(i,temp)
        for (i = 1; i < NN-1; i+=2) {
            if (A[i] > A[i+1]) {
                temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
                exch1=1;
            }
        }
           }
        first=1;
    }}
}
```

Testes e Análise de Resultados

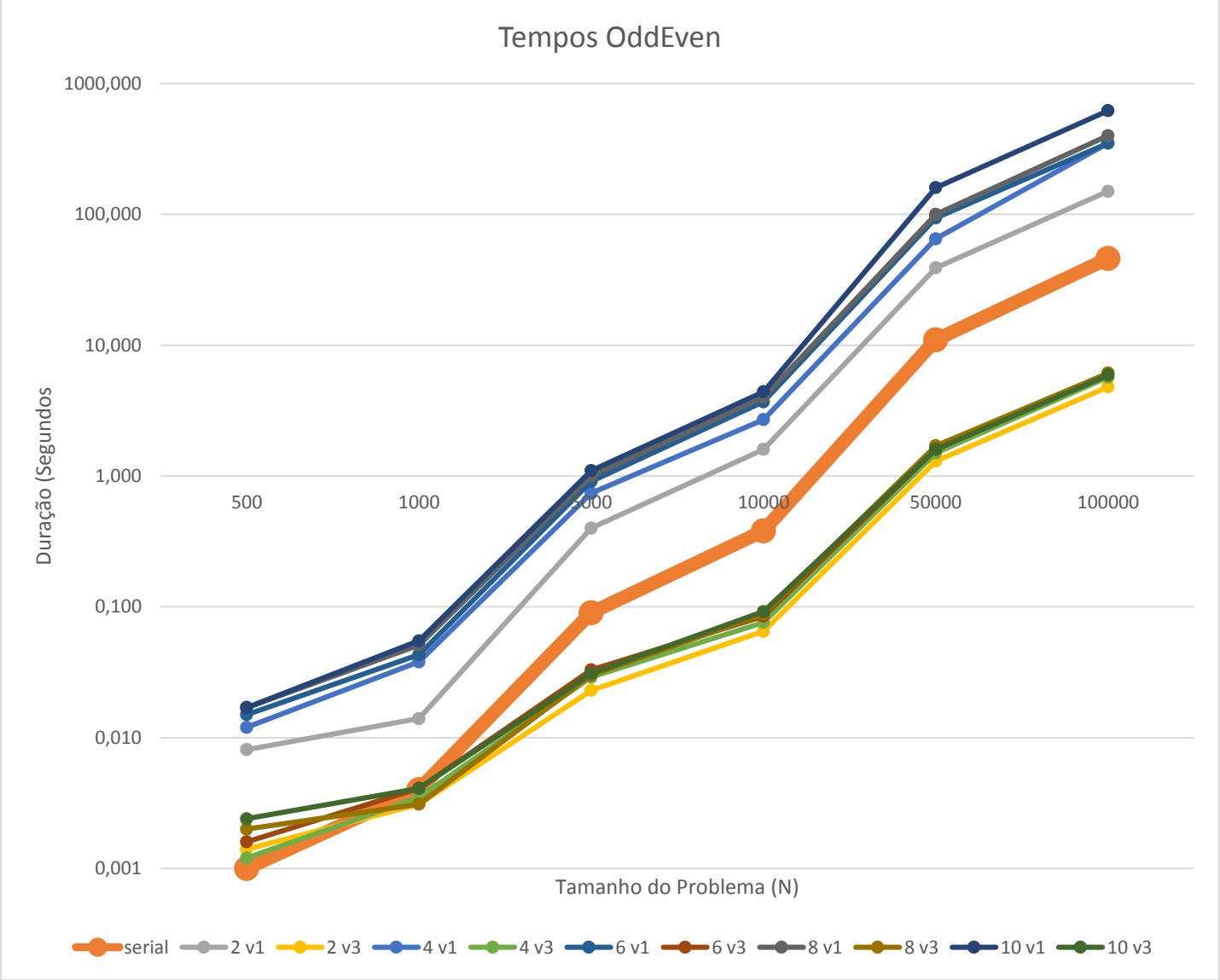
Com tudo concluído, é tempo de fazer medições de tempos de execução e analisar os resultados.

Os testes foram feitos para Dimensões (Size) e número de threads realizados para os 3 tipos de versões. Foram compilados com flag -O2. Os resultados são os seguintes apresentados na tabela.

	OddEven															
	SER	OpenMP														
		2 Threads			4 Threads			6 Threads			8 Threads			10 Threads		
Size	serial	v1	v2	v3	v1	v2	v3	v1	v2	v3	v1	v2	v3	v1	v2	v3
500	0,001	0,008	0,001	0,001	0,012	0,001	0,001	0,015	0,002	0,002	0,017	0,002	0,002	0,017	0,002	0,002
1000	0,004	0,014	0,003	0,003	0,038	0,003	0,004	0,043	0,004	0,004	0,051	0,004	0,003	0,055	0,004	0,004
5000	0,090	0,400	0,023	0,023	0,740	0,028	0,029	0,910	0,028	0,033	1,000	0,033	0,030	1,100	0,032	0,031
10000	0,380	1,600	0,061	0,065	2,700	0,075	0,076	3,700	0,085	0,085	4,000	0,087	0,088	4,400	0,087	0,092
50000	11,0	39,0	1,3	1,3	65,0	1,5	1,5	94,0	1,6	1,6	100,0	1,7	1,7	160,0	1,7	1,6
100000	46,0	150,0	4,3	4,8	350,0	5,7	5,7	350,0	6,1	6,1	400,0	6,2	6,1	620,0	6,1	5,9

Continuando o raciocínio da alínea anterior, em que se queria comparar as versões 2 (abertura e fecho da zona paralela) e 3 (uma zona paralela apenas) que possuem unroll do ciclo for, podemos agora fazer a sua comparação. Como era previsto as diferenças não seriam muitas, conseguindo a versão 3 ficar mais rápida com o aumento de threads e a v2 a diminuir um pouco em comparação com a v3 mas ambas ficam com tempos muito bons. Assim, para problemas de pequena dimensão e recursos recomendo a v2.

Fiquei muito impressionado com os valores obtidos para a v1, sempre conseguiu fazer pior tempo que a serial, a razão direta é devido a todas as barreiras (implícitas ou não) que ele tem conseguindo estar muito tempo nessas zonas. Tal problema foi resolvido com as 2 versões lançadas depois.



Neste gráfico temos uma comparação mais visual dos tempos (foram excluídos os tamanhos 50000 e 100000 pois o tempo da v1 era muito alto e impossibilitava a leitura do mesmo) da versão serial com as diferentes variâncias de threads com a v3 (como os tempos eram muito semelhantes não era necessário ter as duas no gráfico pois iam sobrepor-se frequentemente) e o eixo da Duração em escala logarítmica.

Analisando-o é fácil notar as 4 linhas acima da versão Serial (mais grossa) pois é a versão pior. Depois a ascensão da performance da versão com as duas fases de ciclo a ser melhor que a original com a diferença de threads a não ser muito determinante para estes tamanhos curtos no gráfico.

Conclusão

Para além de ser um caso prático e cativar por si só foi também um trabalho muito bom na medida em que permitiu usar novos conhecimentos adquiridos nesta Unidade Curricular que não foram abordados anteriormente. Um trabalho que apesar de parecer simples engloba em si vários aspectos que tiveram de ser devidamente considerados para que o algoritmo funcionasse corretamente.

É sempre bom conseguir uma versão melhor que a Serial e tal foi obtida usando o unroll do ciclo for em 2 ciclos para tirar partido das threads e diminuir barreiras que são talvez dos fatores que mais tempo fazem perder em versões paralelas. Com esta implementação o número de saídas e entradas da região paralela foi reduzida para metade (v2) e o início dos ciclos for foi definido previamente facilitando o compilador e reduzindo o `pragma omp single` que apenas trocava a variável start. Esta redução de sincronização de threads provou valer a pena ter feito mais código do que inicialmente facultado ficando até mais simples de entender o que faz em relação à versão 1, ou mesmo à serial.

Paralelização OpenMP e PThreads em Multiplicação de Matrizes - Introdução

O problema que nos foi apresentado está relacionado com o algoritmo de multiplicação de Matrizes comparando os tempos e os misses da versão Paralela do PDF, com *OpenMP* e *PThreads*.

Os ficheiros foram compilados com flag -O3 e os Misses simulados com o CacheGrind do Valgrind.

Versão Paralela do PDF

A versão disponibilizada indicou os seguintes resultados:

	Matrix Dimension					
	8,000,000x8		8000x8000		8x8,000,000	
Threads	**Time**	**Eff.**	**Time**	**Eff.**	**Time**	**Eff.**
1	0,322	1	0,264	1	0,333	1
2	0,219	0,735	0,189	0,698	0,3	0,555
4	0,141	0,571	0,119	0,555	0,303	0,275

Versão Paralela OpenMP

Com OpenMP a parte da rotina de multiplicação ficou assim

```
#pragma omp parallel for num_threads(thread_count) default(none) private(i, j)
shared(A, x, y, m, n)
   for (i = 0; i < m; i++){
    y[i] = 0.0;
    for (j = 0; j < n; j++)
    y[i]+= A[i*n+j]*x[j];
   }
```

Testando 3 vezes obtive os seguintes resultados:

Teste 1

	Matrix Dimension					
	8,000,000x8		8000x8000		8x8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0,094	1	0,076	1	0,097	1
2	0,057	0,825	0,049	0,773	0,071	0,683
4	0,039	0,600	0,034	0,555	0,051	0,474

Teste 2

	Matrix Dimension					
	8,000,000x8		8000x8000		8x8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0,093	1	1,000	1	0,095	1
2	0,055	0,857	0,583	0,857	0,070	0,681
4	0,038	0,616	0,406	0,616	0,053	0,445

Teste 3

	Matrix Dimension					
	8,000,000x8		8000x8000		8x8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0,094	1	0,076	1	0,095	1
2	0,057	0,819	0,048	0,798	0,070	0,675
4	0,039	0,599	0,035	0,547	0,051	0,468

Resultando na seguinte média

Teste Média

	Matrix Dimension					
	8,000,000x8		8000x8000		8x8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0,094	1	0,384	1	0,095	1
2	0,056	0,833	0,227	0,847	0,070	0,680
4	0,039	0,604	0,158	0,606	0,052	0,462

Versão Paralela PThreads

Devido ao modo como o PThreads funciona ficou uma versão diferente obrigando a calcular as linhas que cada Thread trabalha. Como se fazia com os processos em MPI.

```
for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
         Pth_mat_vect, (void*) thread);

   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);



...

void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i;
   int j;
   int local_m = m/thread_count;
   register int sub = my_rank*local_m*n;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

#  ifdef DEBUG
   printf("Thread %ld > my_first_row = %d, my_last_row = %d\n",
         my_rank, my_first_row, my_last_row);
#  endif

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
          y[i] += A[sub++]*x[j];
   }

   return NULL;
}
```

Testando 3 vezes obtive os seguintes resultados:

Teste 1

	Matrix Dimension					
	8,000,000x8		8000x8000		8x8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0,085	1	0,054	1	0,036	1
2	0,073	0,579	0,055	0,492	0,028	0,631
4	0,084	0,252	0,073	0,186	0,038	0,236

Teste 2

	Matrix Dimension					
	8,000,000x8		8000x8000		8x8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0,084	1	0,066	1	0,033	1
2	0,091	0,465	0,055	0,600	0,028	0,583
4	0,010	2,108	0,069	0,240	0,038	0,215

Teste 3

	Matrix Dimension					
	8,000,000x8		8000x8000		8x8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0,084	1	0,053	1	0,035	1
2	0,069	0,607	0,046	0,574	0,026	0,662
4	0,085	0,247	0,058	0,231	0,036	0,239

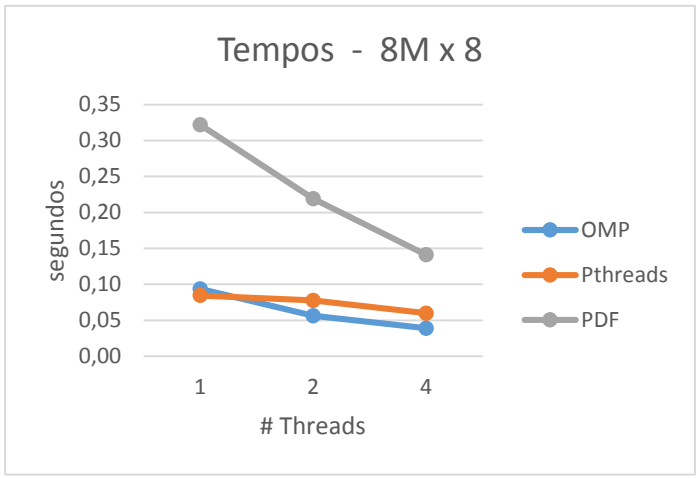
Resultando na seguinte média

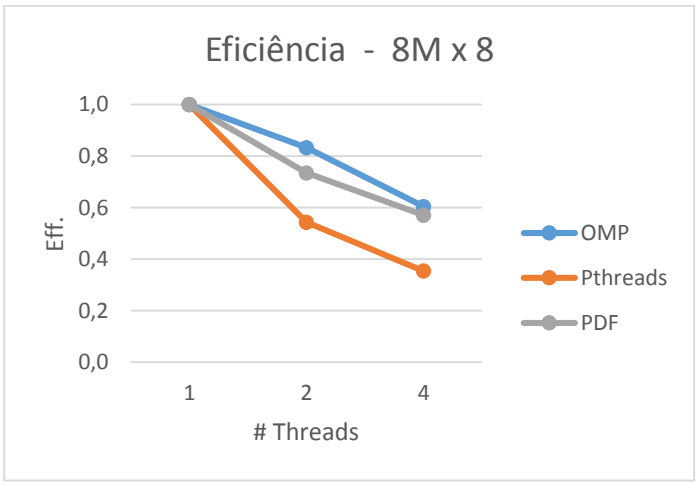
Teste Média

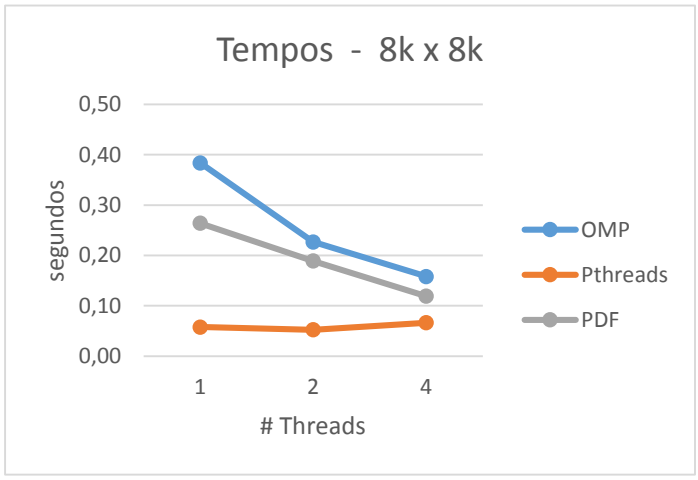
	Matrix Dimension					
	8,000,000x8		8000x8000		8x8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0,084	1	0,058	1	0,034	1
2	0,078	0,544	0,052	0,554	0,028	0,624
4	0,060	0,354	0,066	0,218	0,037	0,230

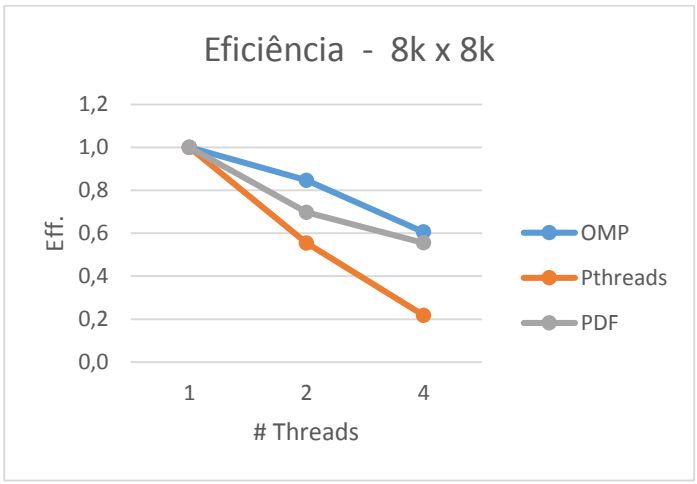
Comparação de Tempos

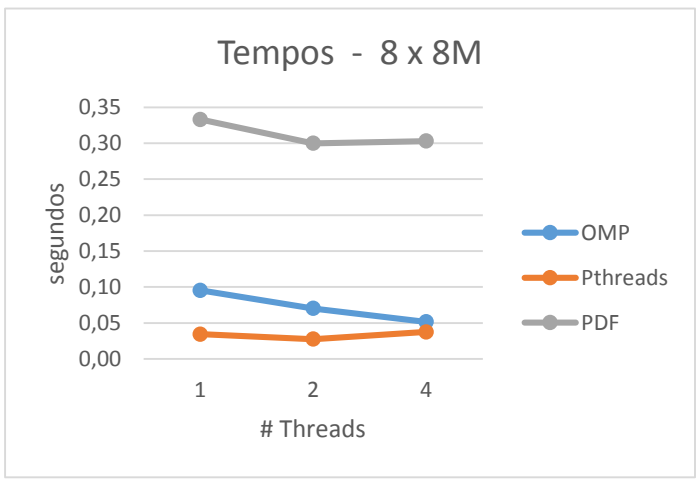
Comparando os Tempos com as 3 implementações, na maior parte das vezes a versão PThreads foi a melhor. Como os Tempos descem mais em relação às outras a eficiência também desce sendo a versão com menor eficiência devido à forma como é calculada.

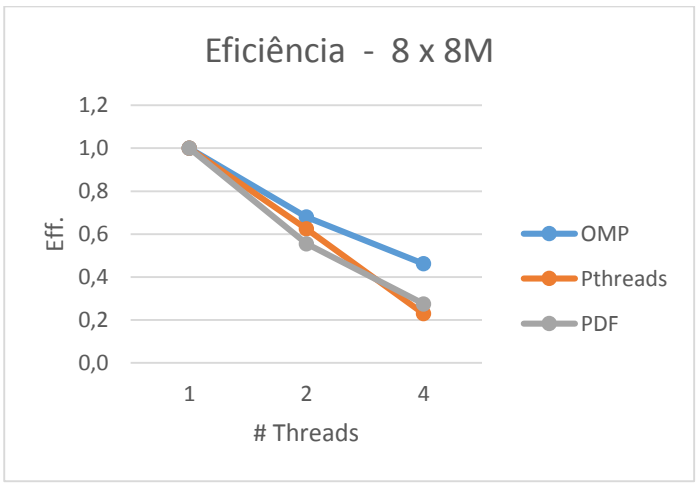






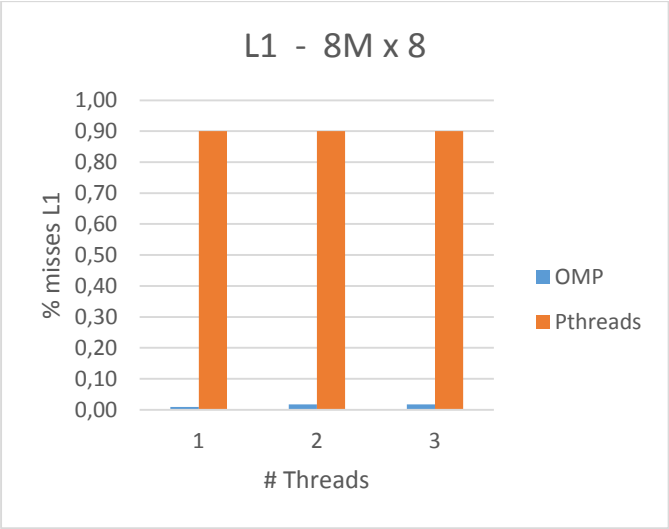


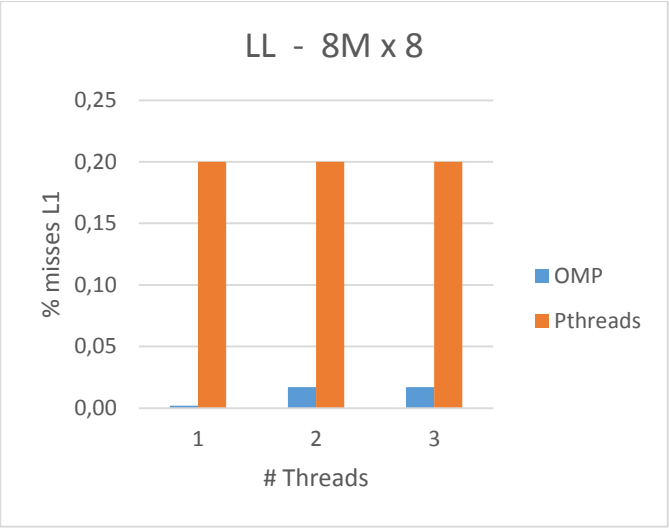


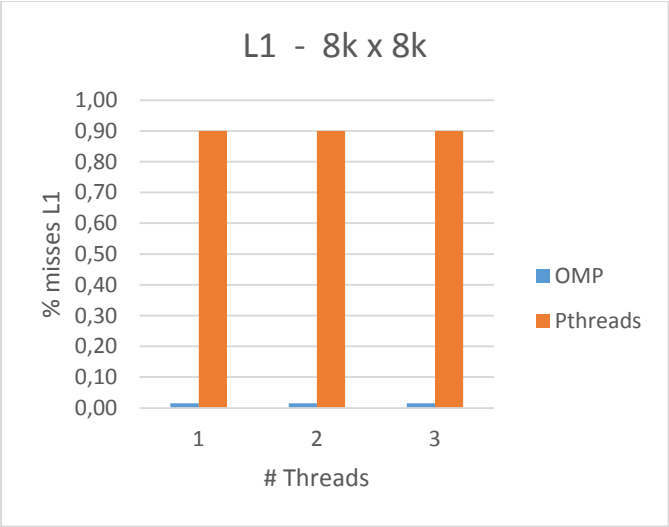


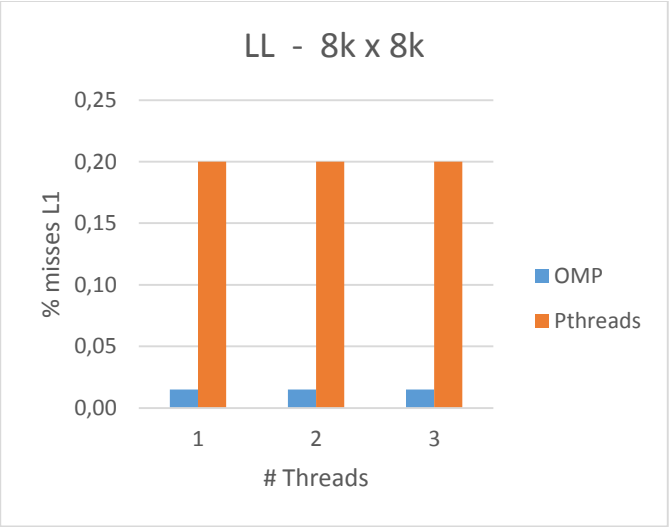
Comparação de Misses

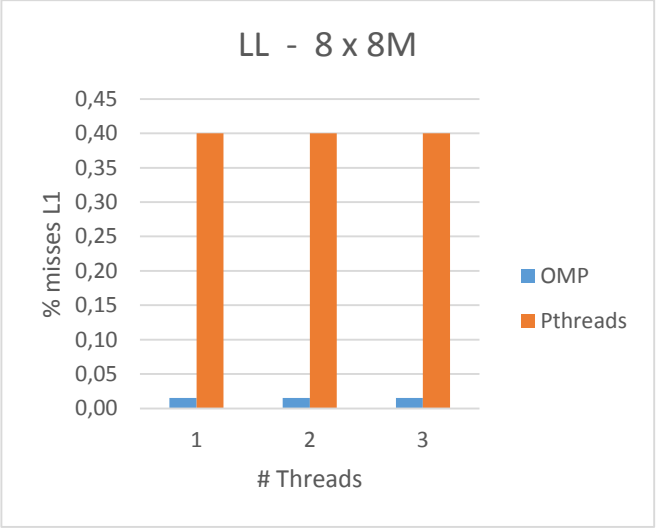
Apesar da versão com PThreads ser a mais rápida, os Misses são muito maiores que a OpenMP.

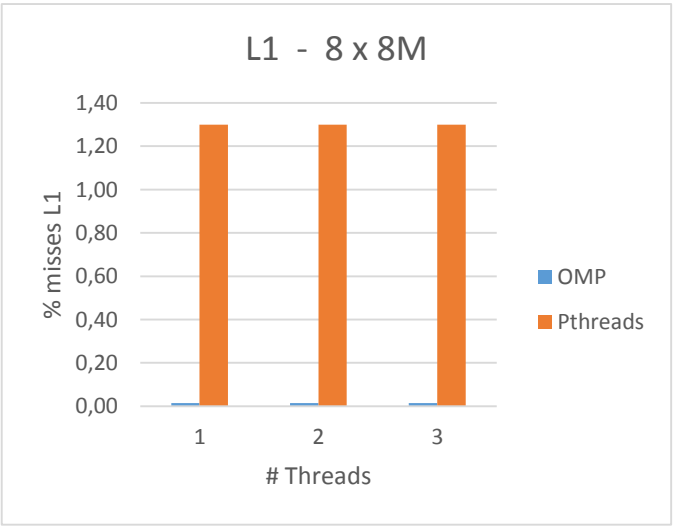












Conclusão

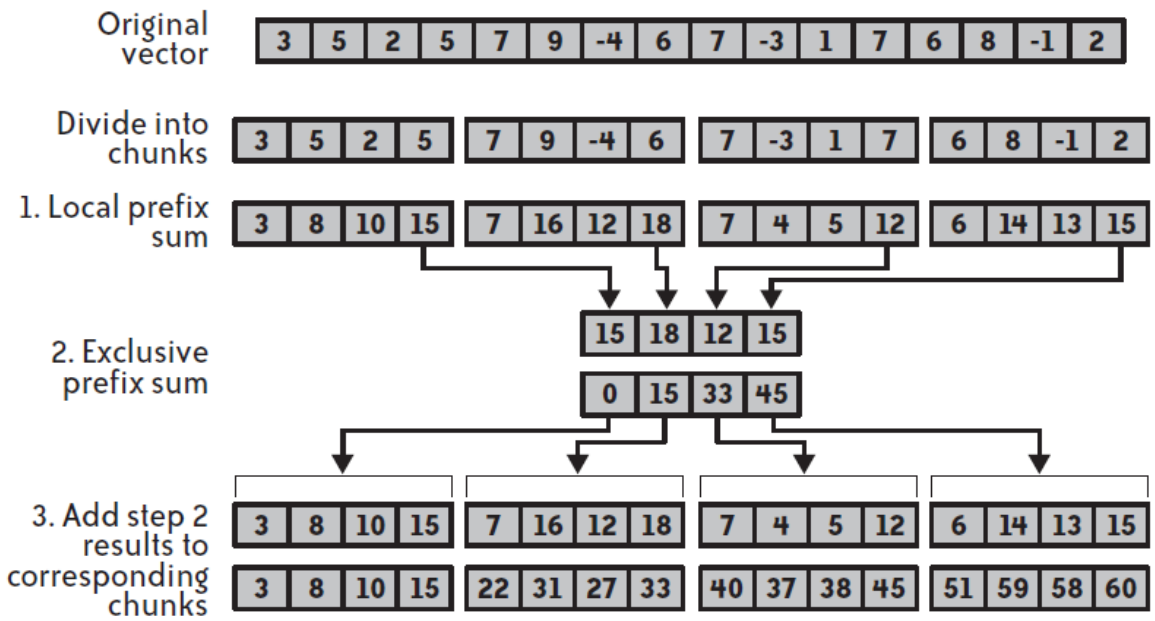
Pela segunda vez consecutiva a versão PThreads conseguiu ser a melhor implementação apesar de ser mais difícil implementa-la pois implica mudanças na estrutura do código enquanto a OpenMP pode ser resolvida com alguns *pragmas* e pequenas alterações apenas no método/função em questão.

Prefix Parallel Sum - Introdução

O problema que nos foi apresentado está relacionado com o algoritmo de Soma Paralela usando Prefix Scan. É nosso trabalho fazer paralelização usando *PThreads* e utilizar *Locks*/*Mutexes*.

Funcionamento do Algoritmo

Iniciando com um vector, neste caso 16 elementos, dividimos em *chunks*. Assim feita a divisão, em cada *chunk* é feita a Prefix Sum e depois é guardado o último elemento de cada num novo *array* de tamanho igual ao número de *chunks*. Com esse array é feito o Exclusive Prefix Sum e com cada elemento é enviado para o chunk respectivo para que seja adicionado a cada elemento dele. Por fim obtemos no *array* inicial em cada elemento, a soma dos elementos anteriores.



Versão Paralela PThreads

Paralelizar este algoritmo implica a alocações dos 2 *arrays inTotals* e *outTotals* com tamanho igual ao número de *Threads* pois será essa a divisão escolhida, 1 *Thread* por *chunk*. Iniciam-se 2 *arrays* de Mutexes de tamanho igual ao número de Threads. O primeiro será para saber quando é que a *Thread* acaba o Prefix no seu chunk e a segunda para ela saber quando é que pode buscar o seu valor ao *array* exclusivo de modo a não termos conflitos. E ainda o *array* de *Threads* usual. É inicializado o *array* e marcado o início do tempo. Os parâmetros número de elementos e de *Threads* é dado por argumento de forma a tornar o programa mais variável.

É criado um ciclo para iniciar as *Threads* e em cada uma é também iniciado o Lock respetivo a ela.

```
int main(int argc, char* argv[])
{
  int j;
  NUM = atoi(argv[1]);
  NUM_THREADS = atoi(argv[2]);

  X = (int*) malloc(sizeof(int)*NUM);
  inTotals = (int*) malloc(sizeof(int)*NUM_THREADS);
  outTotals = (int*) malloc(sizeof(int)*NUM_THREADS);
  mutexs1 = (pthread_mutex_t*) malloc(sizeof(pthread_mutex_t)*NUM_THREADS);
  mutexs2 = (pthread_mutex_t*) malloc(sizeof(pthread_mutex_t)*NUM_THREADS);
  pthread_t * tHandles = (pthread_t*) malloc(sizeof(pthread_t)*NUM_THREADS);

  InitializeArray(X,&NUM);
  double start,end;
  start = omp_get_wtime();

  for (j = 0; j < NUM_THREADS; j++) {
    int *threadNum = (int *)malloc(sizeof (int));
    *threadNum = j;
    pthread_mutex_init(&mutexs1[j], NULL);
    pthread_mutex_init(&mutexs2[j], NULL);
    pthread_mutex_lock (&mutexs1[j]);
    pthread_mutex_lock (&mutexs2[j]);
    pthread_create(&tHandles[j], NULL, Summation, (void *)threadNum);
  }
```

Como veremos a seguir, a *Thread* só liberta o primeiro *mutex* quando acaba o seu Prefix Sum e guardou no *inTotals* o valor do seu último índice. Sendo assim neste segundo ciclo quando o lock é ganho já o podemos destruir pois não faz mais falta e faz-se o Prefix Exclusivo naquela posição específica devido às propriedades mencionadas previamente. É liberto o segundo *mutex* para que a *Thread* respetiva possa continuar sabendo que já tem no *outTotals* o valor calculado.

Depois a Prefix Sum é feita em cada e a escrita do último valor para o novo *array* temporário, chamado de *inTotals* e é feito um lock a um segundo lock.

Por fim é feito o *join* a todas as *Threads* para que o programa saiba que tudo terminou e assim poder terminar, calculando o tempo demorado.

```
for (j = 0; j < NUM_THREADS; j++) {
    pthread_mutex_lock (&mutexs1[j]);
    pthread_mutex_destroy (&mutexs1[j]);
    prefixExclusiveScanPos(inTotals,outTotals,j);
    pthread_mutex_unlock (&mutexs2[j]);
  }

  for (j = 0; j < NUM_THREADS; j++) {
    pthread_join(tHandles[j], NULL);
}

    end = omp_get_wtime();
    printf("%3.2g\n",end-start);

  return 0; }
```

No ponto de vista das *Threads*, é feito o Prefix Scan e preenchido o *inTotals* com o último elemento, que contém a soma de toda a *chunk*, culminando em libertar o primeiro mutex alertando que o seu trabalho do passo 1 está concluído.

Para o próximo passo, o 3, é necessário esperar que seja feita a Exclusive Prefix Sum, passo 2, soma essa feita pelo processo inicial que precisa de todos os valores para a fazer. Só que essa Soma tem uma particularidade, como a soma é feita do índice mais baixo até ao fim e cada posição apenas depende da anterior é possível libertar o valor calculado antes de terminar toda a soma. Ou seja, no exemplo acima, quando é depositado o 15 no *inTotals* (*array* de cima) conseguimos fazer colocar o 0 do *outTotals* (*array* de baixo) e libertar o mutex que entretanto se fez de modo que o passo 3 possa ser feito de seguida. No passo seguinte é a mesma coisa, pegamos no valor anterior do *inTotals*, o 15, somamos ao anterior do *outTotals*, o 0 e depois é libertado o lock para que a segunda *Thread* possa fazer o passo 3. Com esta técnica espero obter melhor performance pois o tempo de espera fica reduzido.

Para saber que a *Thread* já pode usar o valor respetivo no *outTotals*, é feito um lock ao mutex, só que é de esperar que entretanto ele esteja bloqueado até que esse mesmo valor esteja presente. Quando consegue o lock executa o passo 3, que é somar esse valor a todos os elementos do seu *chunk* e de seguida destruir o mutex pois já não será mais preciso e terminar a invocação.

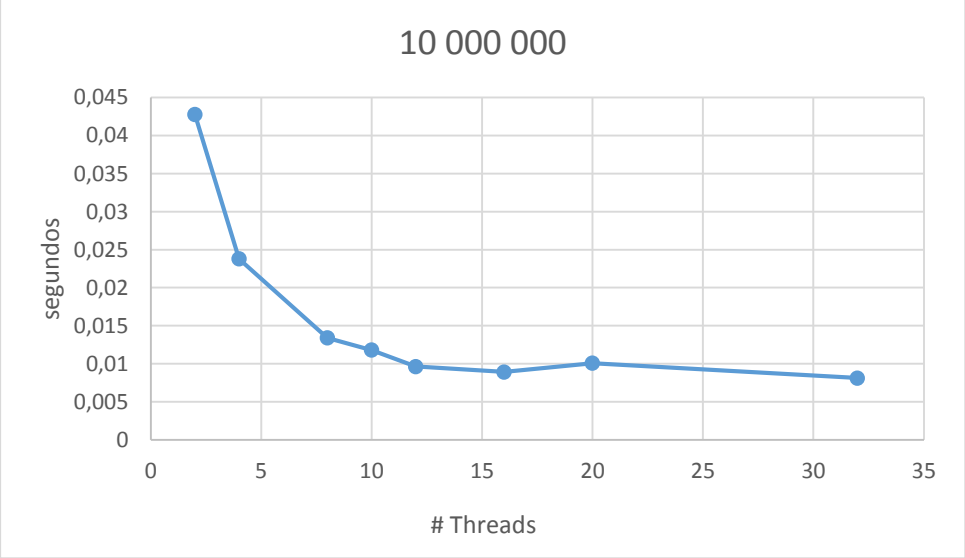
```
void *Summation (void *pArg)
{
  int tNum = *((int *) pArg);
  int lSum = 0;
  int start, end;
  int i,z=0;
  int size = NUM/NUM_THREADS;

  start = (size) * tNum;
  end = (size) * (tNum+1);

  if (tNum == (NUM_THREADS-1)) end = NUM;

  prefixScan(X,start,end);

  inTotals[tNum] = X[end-1];
  pthread_mutex_unlock (&mutexs1[tNum]);
  pthread_mutex_lock (&mutexs2[tNum]);
  for (i = start; i < end; i++)
    {X[i]+=outTotals[tNum];}
  pthread_mutex_destroy (&mutexs2[tNum]);

  delete (int *)pArg;
}



void prefixScan(int * A, int start, int end){
  int i = start+1;
  for(;i<end;i++)
        A[i]+= A[i-1];
}

void prefixExclusiveScanPos(int * A, int * Ord, int pos){
    if(pos==0)
      Ord[0] = 0;
    else
      Ord[pos] = Ord[pos-1]+A[pos-1];
}
```

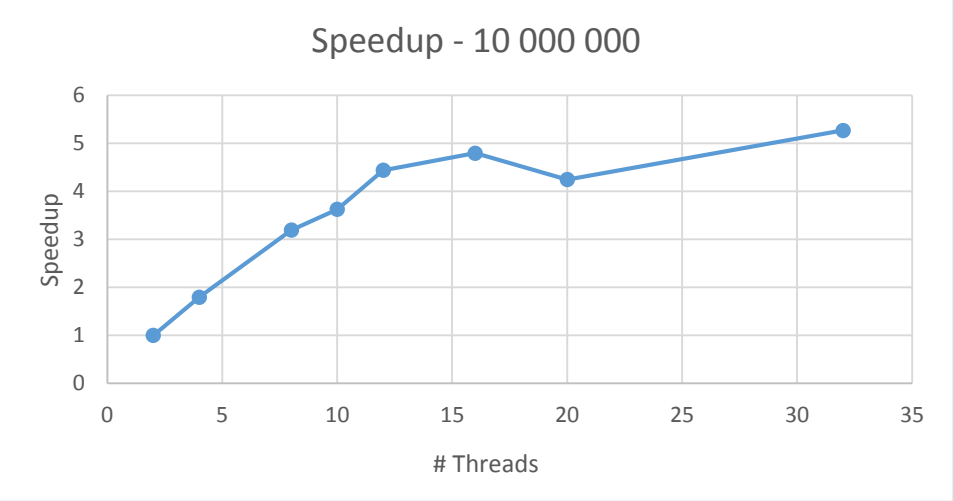
Tempos e Speedup

Foram Realizados 5 testes para cada número de *Threads* e de Elementos, calculada a média desses tempos e feito o *Speedup* baseado no tempo do teste de menor número de *Threads*, 2.

10 Milhões de Elementos

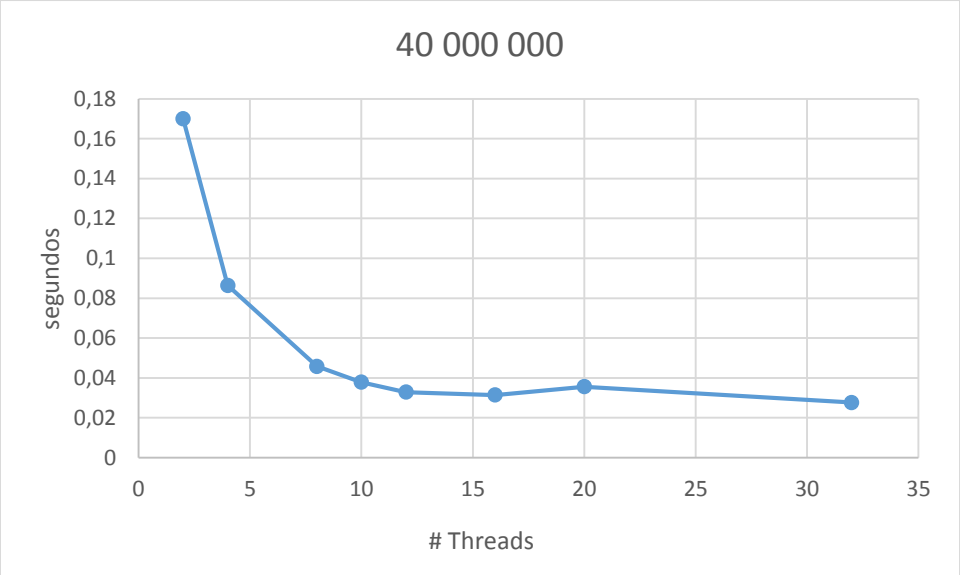
		Tests						
		# 1	# 2	# 3	# 4	# 5	Average	Speedup
Threads	2	0,056	0,04	0,039	0,039	0,04	0,0428	1
	4	0,031	0,022	0,022	0,022	0,022	0,0238	1,798319
	8	0,015	0,013	0,013	0,012	0,014	0,0134	3,19403
	10	0,012	0,012	0,012	0,012	0,011	0,0118	3,627119
	12	0,011	0,0093	0,0094	0,0098	0,0087	0,00964	4,439834
	16	0,011	0,0075	0,0075	0,0076	0,011	0,00892	4,798206
	20	0,011	0,0098	0,01	0,01	0,0096	0,01008	4,246032
	32	0,0099	0,0076	0,0086	0,007	0,0075	0,00812	5,270936

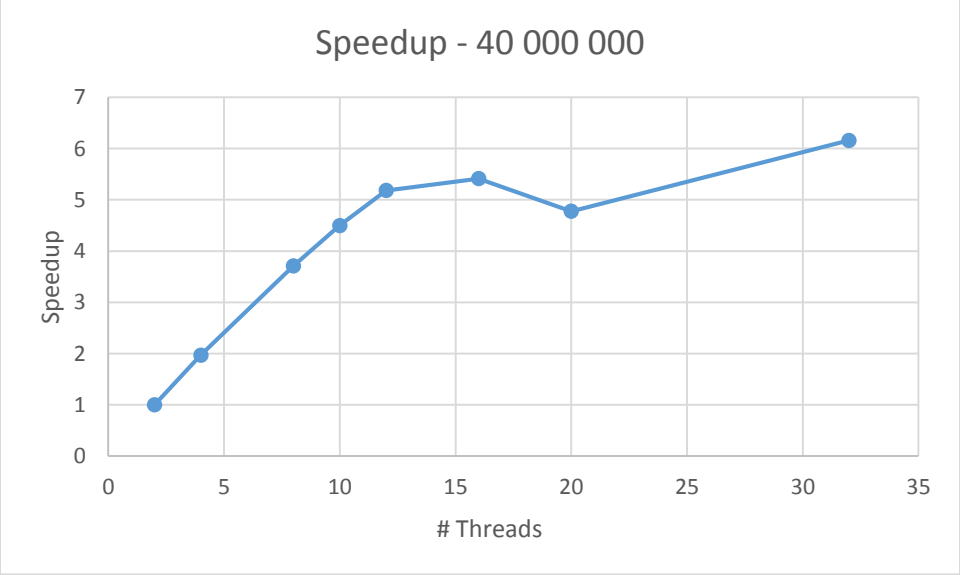




40 Milhões de Elementos

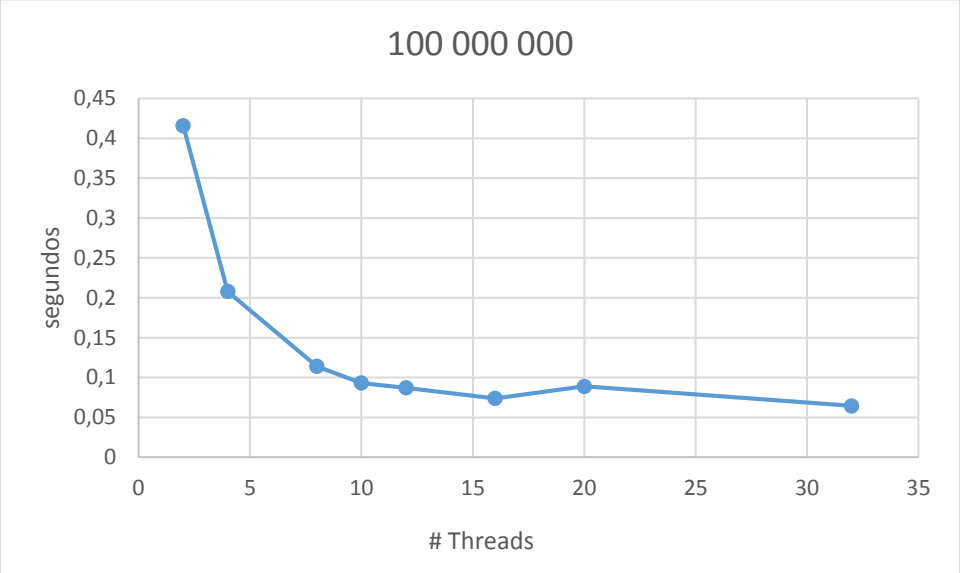
		Tests						
		# 1	**# 2**	**# 3**	**# 4**	**# 5**	**Average**	**Speedup**
Threads	**2**	0,21	0,16	0,16	0,16	0,16	0,17	1
	4	0,096	0,08	0,082	0,087	0,087	0,0864	1,967593
	8	0,045	0,043	0,046	0,048	0,047	0,0458	3,71179
	10	0,036	0,037	0,038	0,039	0,039	0,0378	4,497354
	12	0,033	0,034	0,031	0,033	0,033	0,0328	5,182927
	16	0,038	0,026	0,027	0,026	0,04	0,0314	5,414013
	20	0,041	0,035	0,034	0,035	0,033	0,0356	4,775281
	32	0,03	0,028	0,027	0,027	0,026	0,0276	6,15942

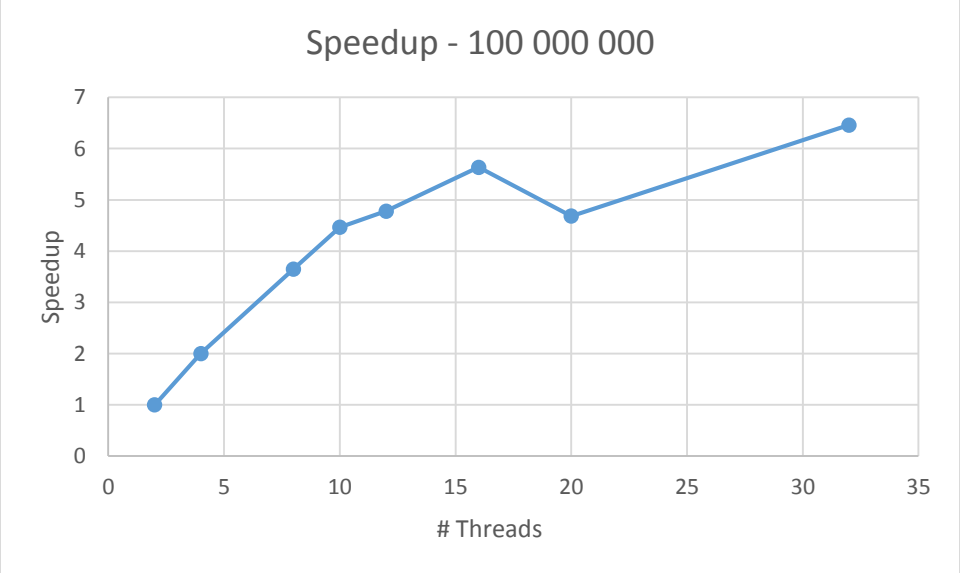




100 Milhões de Elementos

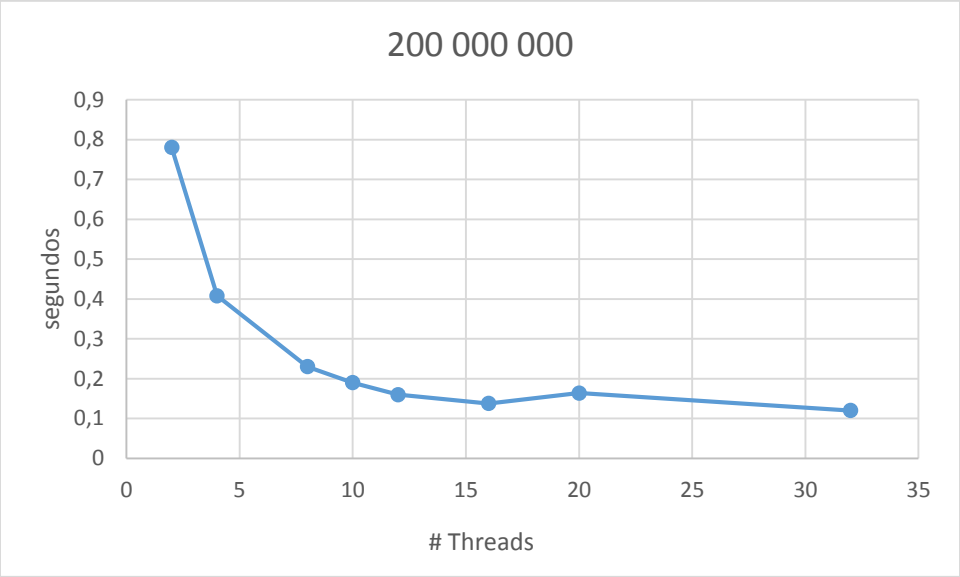
		Tests						
		# 1	# 2	# 3	# 4	# 5	Average	Speedup
Threads	2	0,48	0,43	0,39	0,39	0,39	0,416	1
	4	0,21	0,2	0,2	0,21	0,22	0,208	2
	8	0,11	0,13	0,11	0,11	0,11	0,114	3,649123
	10	0,11	0,09	0,089	0,09	0,087	0,0932	4,463519
	12	0,094	0,095	0,092	0,077	0,077	0,087	4,781609
	16	0,071	0,071	0,07	0,086	0,071	0,0738	5,636856
	20	0,093	0,076	0,096	0,082	0,097	0,0888	4,684685
	32	0,07	0,064	0,068	0,055	0,065	0,0644	6,459627

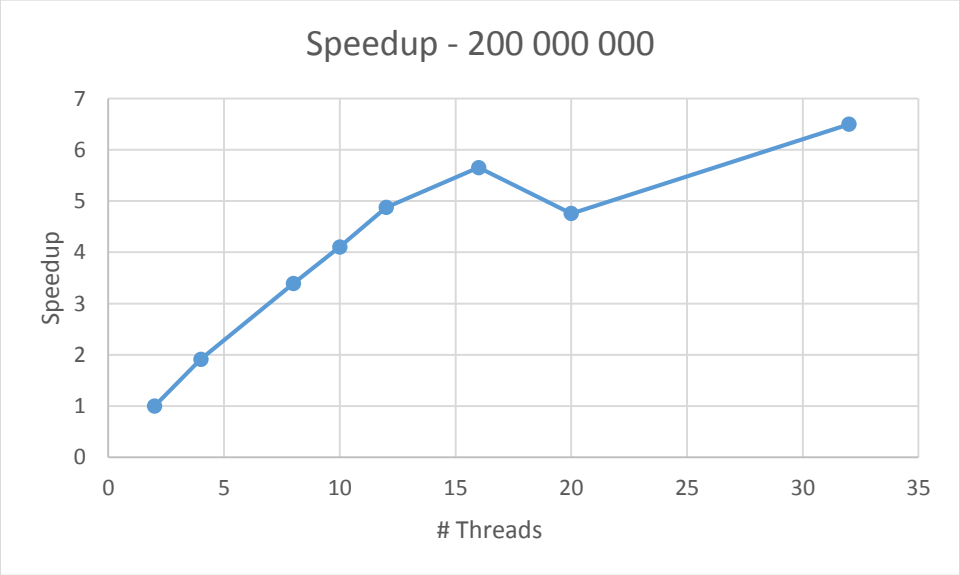




200 Milhões de Elementos

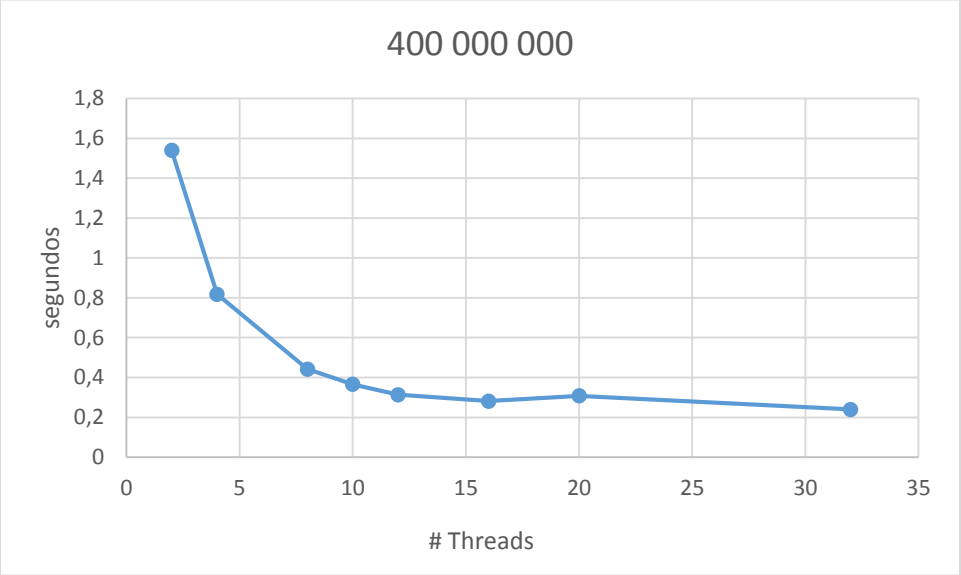
		Tests						
		# 1	**# 2**	**# 3**	**# 4**	**# 5**	**Average**	**Speedup**
Threads	**2**	0,78	0,77	0,8	0,76	0,79	0,78	1
	4	0,4	0,4	0,41	0,41	0,42	0,408	1,911765
	8	0,23	0,23	0,22	0,23	0,24	0,23	3,391304
	10	0,18	0,19	0,2	0,19	0,19	0,19	4,105263
	12	0,17	0,15	0,17	0,16	0,15	0,16	4,875
	16	0,18	0,12	0,13	0,13	0,13	0,138	5,652174
	20	0,16	0,15	0,16	0,17	0,18	0,164	4,756098
	32	0,12	0,12	0,12	0,13	0,11	0,12	6,5

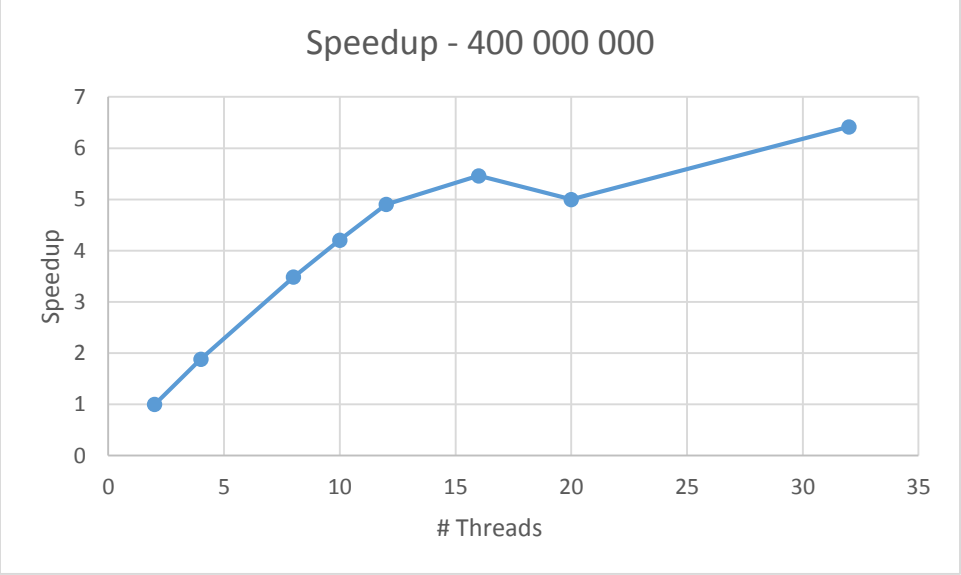




100 Milhões de Elementos

		Tests						
		# 1	**# 2**	**# 3**	**# 4**	**# 5**	**Average**	**Speedup**
Threads	**2**	1,6	1,5	1,6	1,5	1,5	1,54	1
	4	0,82	0,86	0,79	0,8	0,82	0,818	1,882641
	8	0,43	0,44	0,45	0,43	0,46	0,442	3,484163
	10	0,37	0,37	0,38	0,36	0,35	0,366	4,20765
	12	0,31	0,31	0,31	0,32	0,32	0,314	4,904459
	16	0,31	0,24	0,24	0,27	0,35	0,282	5,460993
	20	0,3	0,3	0,32	0,3	0,32	0,308	5
	32	0,24	0,24	0,25	0,23	0,24	0,24	6,416667





Conclusão e Análise de Resultados

Os valores, obtidos no *compute-541-1* com job de 32 *Threads*, de Speedup são muito semelhantes entre os vários tamanhos de input, tal que o algoritmo escala relativamente bem. Por exemplo para 32 Threads obtive um speedup à volta de 6, ou seja, é 6x mais rápido a executar o programa. Os ganhos são muito bons.

Quanto ao trabalho em si, era clara a necessidade de utilização de Locks na implementação de *PThreads*. Optei por uma abordagem mais radical tentando reduzir o tempo de espera nos *locks*, portanto tive que utilizar *Mutexes* exclusivos. A abordagem mais fácil e direta seria usando um Semáforo, criando uma espécie de Barreira (típica do *OpenMP*) para que quando todos os *chunks* estivessem contabilizados no passo 1, as Threads respectivas ficariam em espera para que o processo principal fizesse o Exclusive Sum em paz e quando terminasse dava luz verde ao Semáforo e todas as *Threads* finalizavam. Achei que esse tempo de espera era inútil pois a primeira *Thread* não precisa de esperar até que tudo termine, pode seguir logo o seu caminho mal o valor esteja determinado.

MapReduce com MPI - Introdução

O problema que nos foi apresentado trata-se de implementar 2 versões do Friendly Numbers em Map Reduce com MPI e com MRMPI. Depois comparar os resultados.

Funcionamento do Algoritmo

Tendo como objetivo encontrar os números amigáveis de `start` a `end`, o algoritmo original está em OpenMP para que se possa tirar partido do paralelismo. Para comparar se 2 números são amigáveis, têm que ter o numerador e o denominador iguais, por isso temos que os guardar para futura comparação. Portanto é guardado um array `num`, `den` e `the_num` para sabermos de que números se tratam.

Cada threads para o seu número guarda nos arrays os valores calculados. No fim é feita uma comparação um a um para encontrar pares `num`/`den` iguais.

```
void FriendlyNumbers (int start, int end)
{
    int last = end-start+1;
    int *the_num = new int[last];
    int *num = new int[last];
    int *den = new int[last];
#pragma omp parallel
 {  int i, j, factor, ii, sum, done, n;
    // -- MAP --
    #pragma omp for schedule (dynamic, 16)
    for (i = start; i <= end; i++) {
        ii = i - start;
        sum = 1 + i;
        the_num[ii] = i;
        done = i;
        factor = 2;
        while (factor < done) {
            if ((i % factor) == 0) {
                sum += (factor + (i/factor));
                if ((done = i/factor) == factor) sum -= factor;
            }
            factor++;
        }
        num[ii] = sum; den[ii] = i;
        n = gcd(num[ii], den[ii]);
        num[ii] /= n;
        den[ii] /= n;
    } // end for
// -- REDUCE --
#pragma omp for schedule (static, 8)
    for (i = 0; i < last; i++) {
        for (j = i+1; j< last; j++) {
            if ((num[i] == num[j]) && (den[i] == den[j]))
                printf ("%d and %d are FRIENDLY \n", the_num[i], the_num[j]);
        }
    }
 } // end parallel region
    delete[] the_num;
    delete[] num;
    delete[] den;
}
```

Versão MPI simples

A Paralelização com MPI é feita dividindo um range de valores para cada processo. Depois invocar a `FriendlyNumbers` com esse range e no fim enviar para o processo 0 todos os dados calculados, portanto o pid faz também trabalho computacional mas irá depois receber a informação de todos e calcular os números amigáveis.

```
int main(int argc, char **argv)
{
    unsigned int start = atoi(argv[1]), end = atoi(argv[2]);
    MPI_Init(&argc,&argv);
    double time=MPI_Wtime();    int pid,n_proc;
    MPI_Status status1,status2;
    MPI_Comm_size(MPI_COMM_WORLD,&n_proc); MPI_Comm_rank(MPI_COMM_WORLD,&pid);
    size = end-start+1;
    int offset = size / (n_proc);
    int off = 0,proc;

    printf("pid:%d  start:%d end:%d offset:%d size:%d\n",pid,start+(offset*pid),
start+(offset*(pid+1)),offset,size);
    if(pid==0){
      num = new int[size]; den = new int[size];}
    else
    {
      num = new int[offset]; den = new int[offset];}

        FriendlyNumbers(start+(offset*(pid)), start+(offset*(pid+1)-1));
    if(pid==0){
        int i=1,j;
        for(i;i<n_proc;i++){
        MPI_Recv(&num[(i)*offset],offset, MPI_INT, i,1,MPI_COMM_WORLD,&status1);
        MPI_Recv(&den[(i)*offset],offset, MPI_INT, i,2,MPI_COMM_WORLD,&status2); }

// -- REDUCE --
#pragma parallel omp for schedule (static, 8)
    for (i = 0; i < size; i++)
        for (j = i+1; j< size; j++)
            if ((num[i] == num[j]) && (den[i] == den[j]))
                printf ("%d and %d are FRIENDLY \n", start+i, start+j);

            time=MPI_Wtime()-time;
        printf("Time:%f seconds\n");
    }
    else{
    MPI_Send(&num[0],offset, MPI_INT, 0,1,MPI_COMM_WORLD);
    MPI_Send(&den[0],offset, MPI_INT, 0,2,MPI_COMM_WORLD);
  }
    MPI_Finalize();
}
```

Versão MapReduceMPI

Foi necessário introduzir a *struct* `Pair` que vai ser a Chave do *Map*, é constituída por 2 inteiros, que serão correspondidos ao numerador e denominador. A `charTointStar` vai passar um array de *char's* para inteiros para podermos ler os números amigáveis, correspondentes ao Valor do *Map*. A *output* trata do *reduce*, pega nas chaves que tenham mais que um valor (ou seja têm pelo menos um par de números amigáveis) e faz o print dos números.

```
typedef struct sPair {
  int x, y;
}Pair;



void charTointStar(int* ints, char*multivalue, int nvalues,int* vb)
{
    int* values = (int*) multivalue;
    for (int i = 0; i < nvalues; ++i)
    {
        ints [i] = values[i];
    }
}


void output(char *key, int keybytes, char *multivalue, int nvalues, int
*valuebytes, KeyValue *kv, void *ptr)
{
    int* numbers=(int*)malloc(sizeof(int)*nvalues);

    charTointStar(numbers,multivalue,nvalues,valuebytes);
    if(nvalues>1)
    {
        for (int i = 0; i < nvalues; i++)
        {
           printf("%d ", multivalue[i]);
        }
        printf("\n");
    }
`
```

A `FriendlyNumbers` é a função que trata do *Map*. É criado o *Pair p* que será a chave e adicionado o Valor com o número atual.

```
void FriendlyNumbers(int itask, KeyValue *kv, void* ptr)
{

    int num,den;
    #pragma omp parallel
   {    int i, j, factor, ii, sum, done, n;
    // -- MAP --
    #pragma omp for schedule (dynamic, 16)
    for (i = start; i <= end; i++) {
        ii = i - start;
        sum = 1 + i;
        done = i;
        factor = 2;
        while (factor < done) {
            if ((i % factor) == 0) {
                sum += (factor + (i/factor));
                if ((done = i/factor) == factor) sum -= factor;
            }
            factor++;
        }
        num = sum;
        den = i;
        n = gcd(num, den);
        num /= n;
        den /= n;

        Pair p;p.x=num;p.y=den;
        int val = i;
        kv->add((char*)&p,sizeof(Pair),(char*) &val,sizeof(int));

    } // end for
 }
  // end parallel region
}
```

No main, instancia-se o MapReduce, vem a barreira, é iniciado o map com o `FriendlyNumbers`, é feito o *collate* e de seguida o *reduce* com o output.

```
// MapReduce
   MapReduce *mr = new MapReduce(MPI_COMM_WORLD);
   mr->verbosity=0;
   mr->timer = 1;
   MPI_Barrier(MPI_COMM_WORLD);
   int nNumber = mr->map(n_proc,&FriendlyNumbers,NULL);
   int nChunks = mr->mapfilecount;
   mr->collate(NULL); // Collate Keys

   if(pid==0)
        printf("The following numbers are friendly:\n");
   int nunique = mr->reduce(output,NULL);
   MPI_Barrier(MPI_COMM_WORLD);
```

Tempos e Speedup

Foram Realizados 5 testes para cada *Size* e número de Processos, calculada a média desses tempos e feito o *Speedup* baseado no tempo da versão MPI.

Tabela Comparativa

			Size				Speedups			
			1000	10000	1000000	10000000	1000	10000	1000000	10000000
Procs	2	MPI	0,000648	0,031002	2,361124	192,8277	48,81481	2,027772	1,026414	1,000801
		MRMPI	0,031632	0,062865	2,423491	192,9821				
	4	MPI	0,000367	0,018001	1,359308	111,2274	110,7711	3,269429	1,039153	1,002239
		MRMPI	0,040653	0,058853	1,412529	111,4764				
	8	MPI	0,0008	0,009842	0,730489	59,52372	89,80625	8,273217	1,116176	1,004068
		MRMPI	0,071845	0,081425	0,815354	59,76585				
	10	MPI	0,000709	0,008303	0,589432	47,96692	106,2003	9,78297	1,140802	1,004857
		MRMPI	0,075296	0,081228	0,672425	48,19988				
	12	MPI	0,041271	0,047512	0,535457	40,31815	2,855274	2,636155	1,154401	1,004971
		MRMPI	0,11784	0,125249	0,618132	40,51857				
	16	MPI	0,00182	0,005867	0,375138	30,62136	45,32912	14,77808	1,256407	1,015962
		MRMPI	0,082499	0,086703	0,471326	31,11015				
	20	MPI	0,001776	0,00462	0,302531	24,51627	49,09685	19,35844	1,313492	1,055852
		MRMPI	0,087196	0,089436	0,397372	25,88555				

A versão em MPI foi sempre melhor que a MRMPI mas com o aumento do Size a diferença diluiu-se.

mpiP para versão MPI

Como podemos analisar pelo resultado obtido da execução com *mpiP*, quanto maior o *Size* maior o tempo que é preciso esperar para receber os valores no primeiro nó, porque o cálculo do maior divisor comum é mais lento para valores altos e o *pid 0* é o que acaba mais rápido. Para valores baixos a diferença ainda é pouca. Para 4 processos.

	Call	Site	Time	App%	MPI%	COV
1000	**Recv**	1	0.016	1.06	2,09	0.00
	Recv	2	0.337	22.24	44,11	0.00
	Send	3	0.024	1.58	3,14	0.57
	Send	4	0.387	25.54	50,65	0.17
100000	**Recv**	1	0.353	0.01	0,03	0.00
	Recv	2	1.12e+03	25.75	99,85	0.00
	Send	3	0.344	0.01	0,03	0.27
	Send	4	0.936	0.02	0,08	0.13

mpiP para versão MRMPI

Era de esperar que para valores baixos a Barreira introduzida para sincronismo fosse trazer problemas e é o que o *mpiP* confirmou. Depois como seria de esperar a Allreduce tem muita carga computacional, o mrMPI tem que enviar para todos os processos os valores para fazerem reduce. Para 4 processos.

	Call	Site	Time	App%	MPI%	COV
1000	Barrier	3	0,803	0,5	14,19	0,2
	Allreduce	50	0,405	0,25	7,16	0
	Allreduce	25	0,328	0,21	5,8	0
	Allreduce	7	0,288	0,18	5,09	0
	Allreduce	16	0,272	0,17	4,81	0
	Allreduce	67	0,256	0,16	4,52	0
	Allreduce	42	0,217	0,14	3,84	0
	Allreduce	20	0,209	0,13	3,69	0
	Barrier	56	0,142	0,09	2,51	0
	Barrier	39	0,142	0,09	2,51	0
	Barrier	22	0,135	0,08	2,39	0
	Allreduce	69	0,132	0,08	2,33	0
	Allreduce	33	0,12	0,08	2,12	0
	Barrier	31	0,112	0,07	1,98	0
	Allreduce	52	0,102	0,06	1,8	0
	Alltoall	37	0,096	0,06	1,7	0
	Alltoall	54	0,096	0,06	1,7	0
	Alltoall	21	0,094	0,06	1,66	0
	Alltoall	2	0,093	0,06	1,64	0
	Allreduce	55	0,089	0,06	1,57	0

	Call	Site	Time	App%	MPI%	COV
100000	Allreduce	16	1,12E+03	19,73	50,42	0
	Allreduce	50	725	12,77	32,63	0
	Allreduce	33	356	6,26	16	0
	Alltoallv	63	3,14	0,06	0,14	0
	Alltoallv	46	3,1	0,05	0,14	0
	Alltoallv	29	2,64	0,05	0,12	0
	Alltoallv	11	1,73	0,03	0,08	0
	Allreduce	64	1,69	0,03	0,08	0
	Allreduce	47	1,66	0,03	0,07	0
	Allreduce	30	1,64	0,03	0,07	0
	Barrier	3	0,715	0,01	0,03	0,08
	Allreduce	53	0,598	0,01	0,03	0
	Allreduce	70	0,566	0,01	0,03	0
	Allreduce	36	0,462	0,01	0,02	0
	Allreduce	18	0,309	0,01	0,01	0
	Barrier	13	0,235	0	0,01	0
	Allreduce	7	0,228	0	0,01	0
	Allreduce	59	0,179	0	0,01	0
	Allreduce	1	0,169	0	0,01	0
	Allreduce	55	0,133	0	0,01	0

Conclusão e Análise de Resultados

Os valores, obtidos no *compute-321-1* com job de 20 *processos*, de Speedup são muito semelhantes à medida que o Size aumenta. Para valores baixos a utilização do MapReduce não é recomendável.