Universidade do Minho

Mestrado em Engenharia Informática

Algoritmos Paralelos - 2014/2015

Prefix Parallel Sum

18 de Maio de 2015

Fábio Gomes pg27752

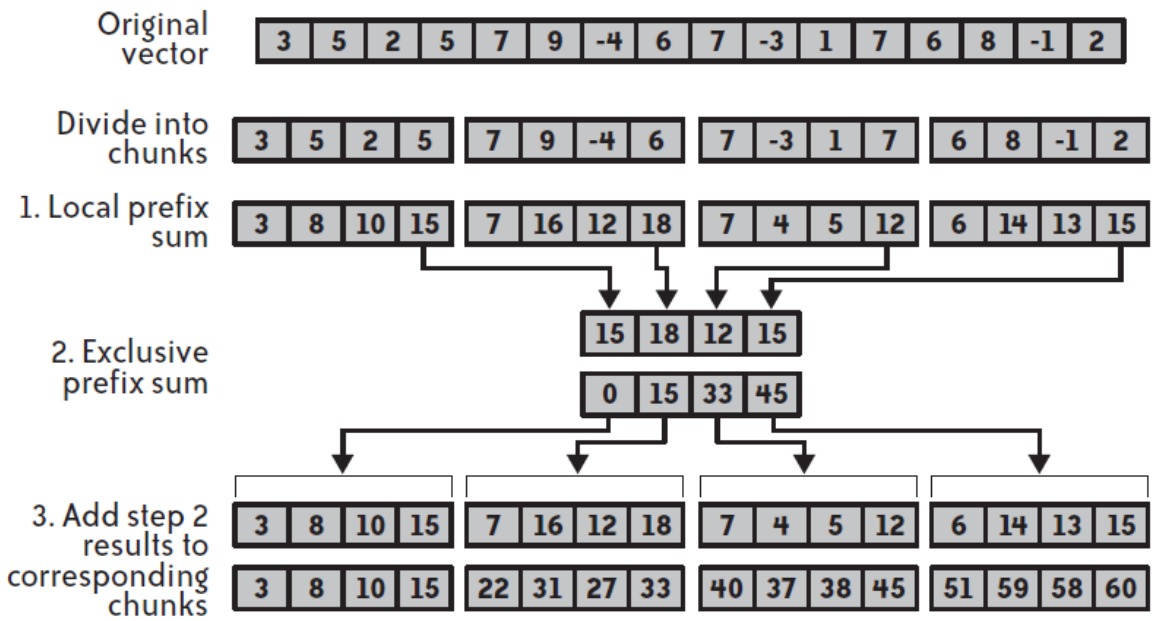
Índice

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.

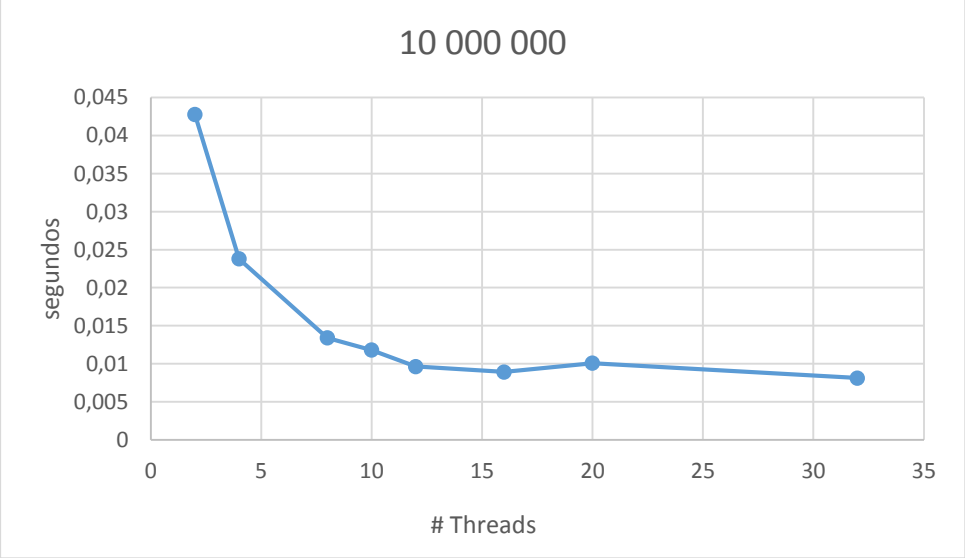
```
void *Summation (void *pArg)
{
  int tNum = *((int *) pArg);
  int lSum = 0;
  int start, end;
  int i,z=0;
  int size = NUM/NUM_THREADS;

  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];
}
```

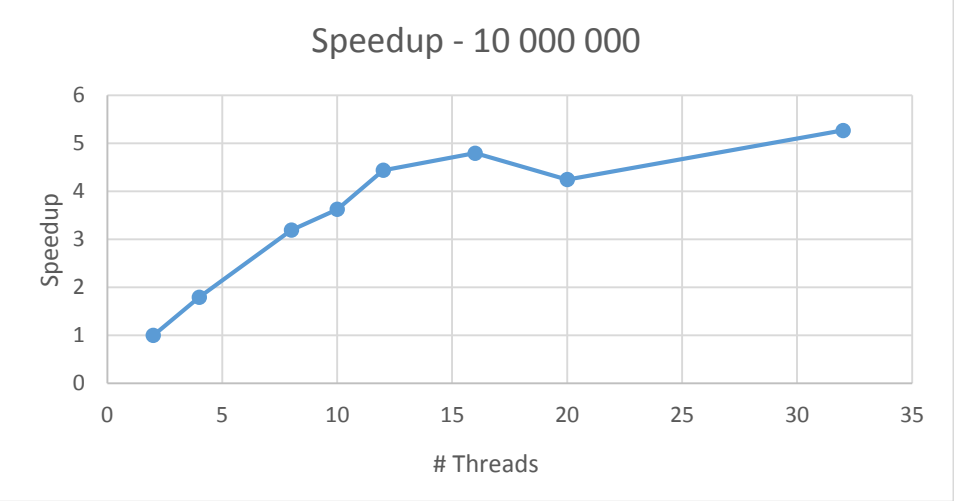
Tempos e Speedup

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

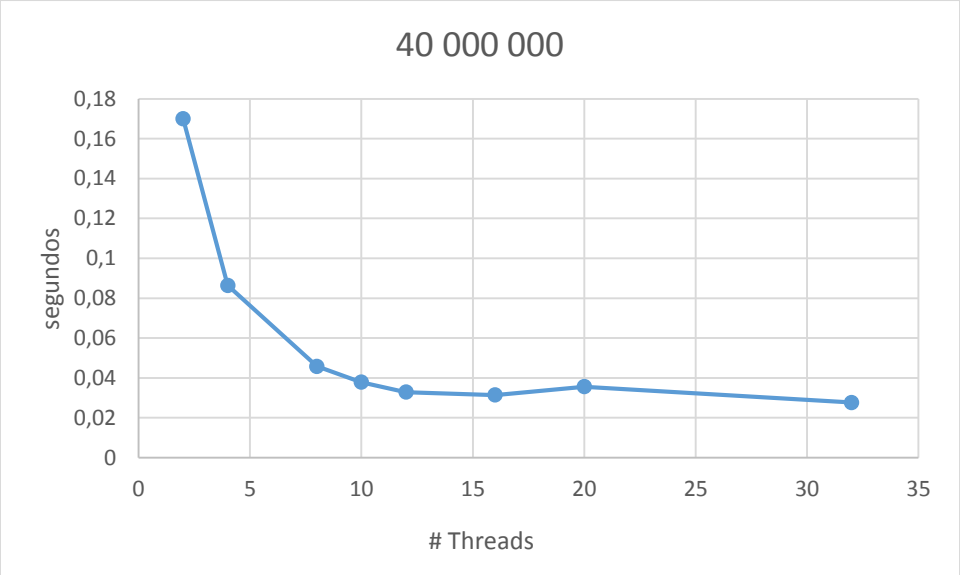
		Tests						
		# 1	# 2	# 3	# 4	# 5	Average	Speedup
Threads	2	0,056	0,04	0,039	0,039	0,04	0,0428	1
	4	0,031	0,022	0,022	0,022	0,022	0,0238	1,798319
	8	0,015	0,013	0,013	0,012	0,014	0,0134	3,19403
	10	0,012	0,012	0,012	0,012	0,011	0,0118	3,627119
	12	0,011	0,0093	0,0094	0,0098	0,0087	0,00964	4,439834
	16	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

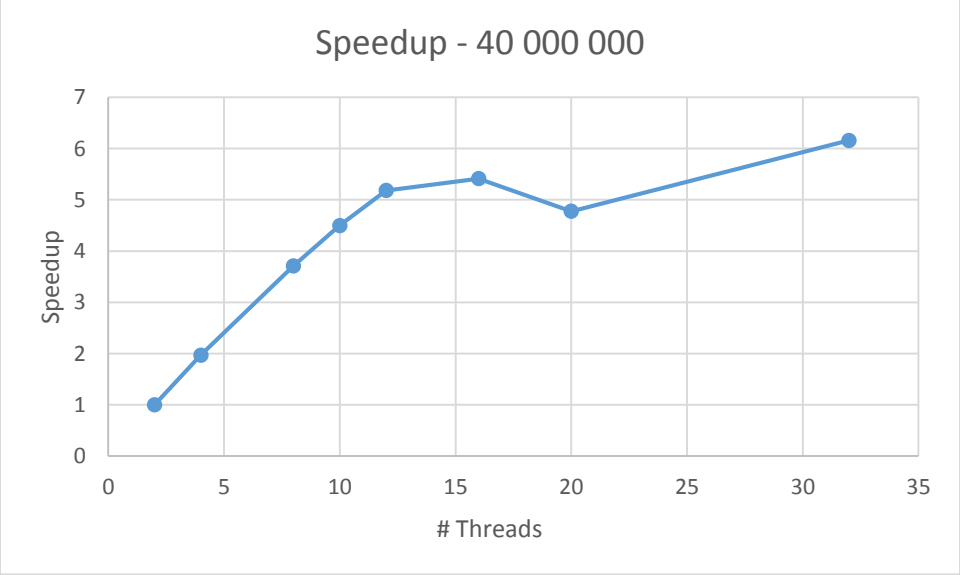




40 Milhões de Elementos

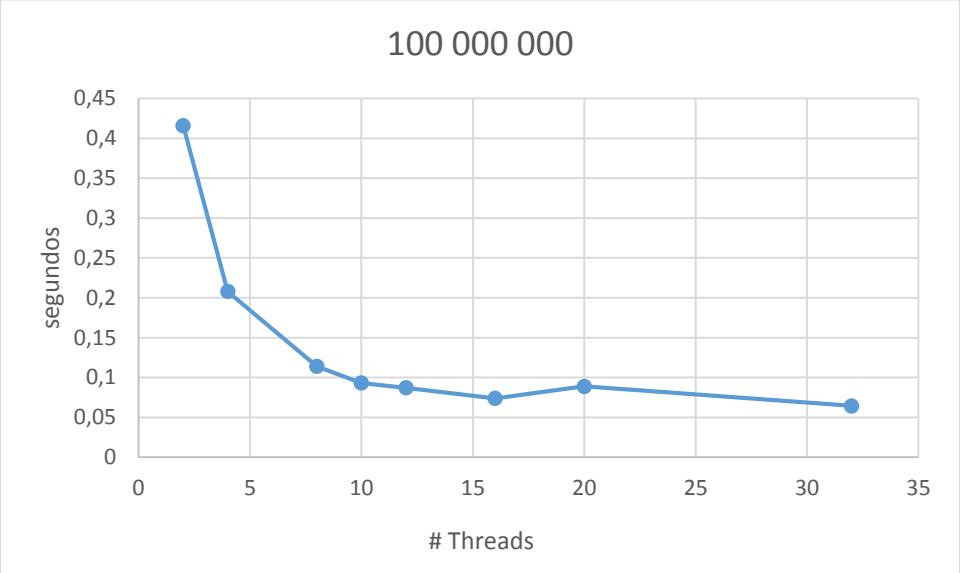
		Tests						
		# 1	**# 2**	**# 3**	**# 4**	**# 5**	**Average**	**Speedup**
Threads	**2**	0,21	0,16	0,16	0,16	0,16	0,17	1
	4	0,096	0,08	0,082	0,087	0,087	0,0864	1,967593
	8	0,045	0,043	0,046	0,048	0,047	0,0458	3,71179
	10	0,036	0,037	0,038	0,039	0,039	0,0378	4,497354
	12	0,033	0,034	0,031	0,033	0,033	0,0328	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

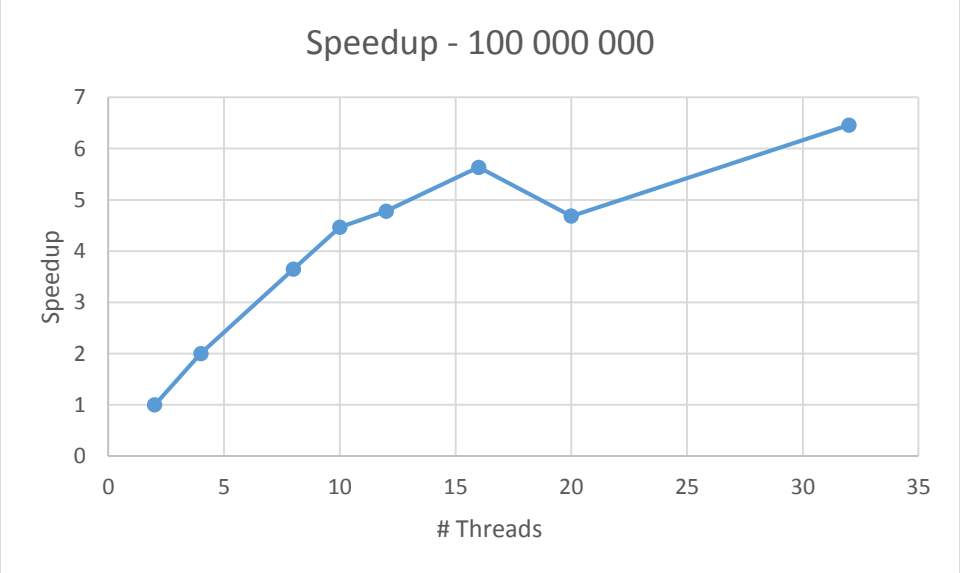




100 Milhões de Elementos

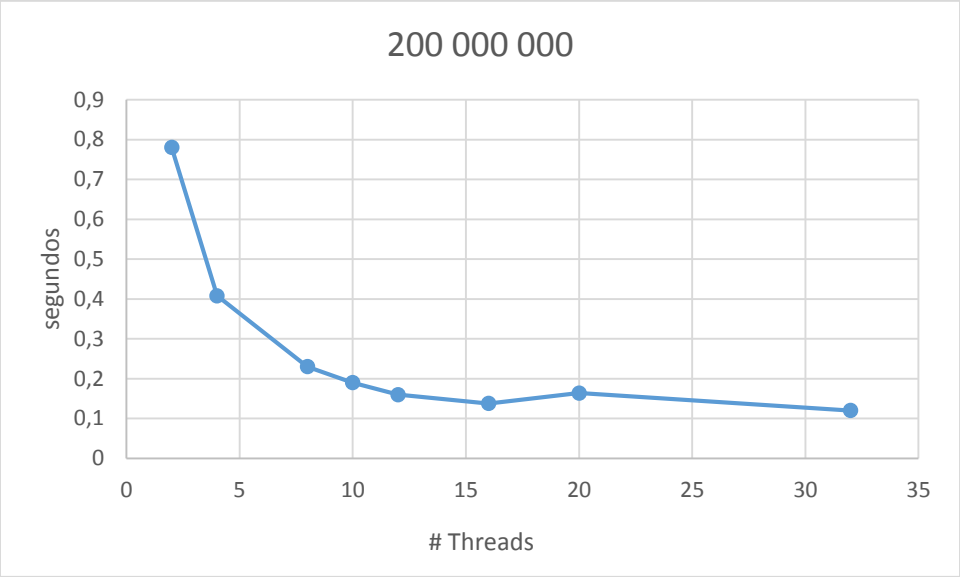
		Tests						
		# 1	# 2	# 3	# 4	# 5	Average	Speedup
Threads	2	0,48	0,43	0,39	0,39	0,39	0,416	1
	4	0,21	0,2	0,2	0,21	0,22	0,208	2
	8	0,11	0,13	0,11	0,11	0,11	0,114	3,649123
	10	0,11	0,09	0,089	0,09	0,087	0,0932	4,463519
	12	0,094	0,095	0,092	0,077	0,077	0,087	4,781609
	16	0,071	0,071	0,07	0,086	0,071	0,0738	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

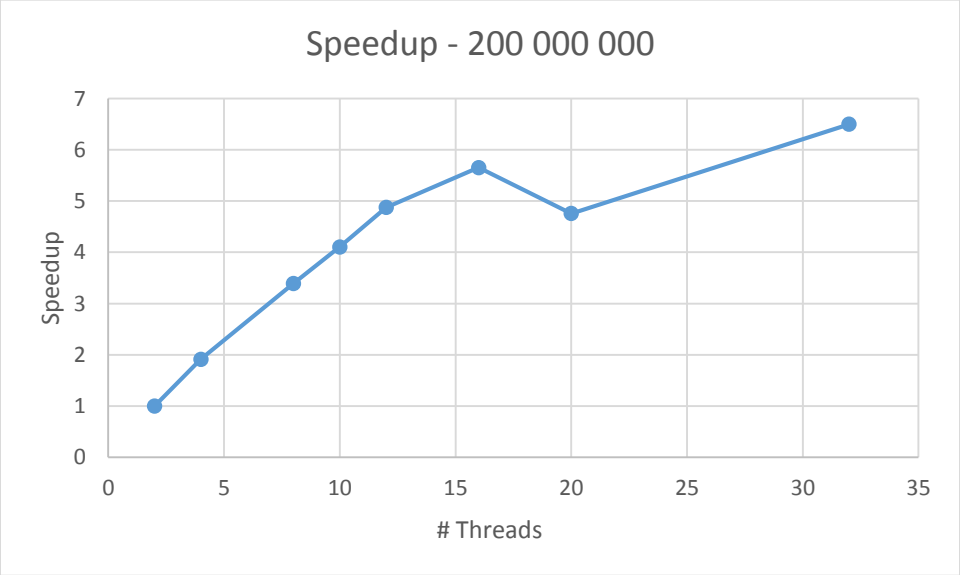




200 Milhões de Elementos

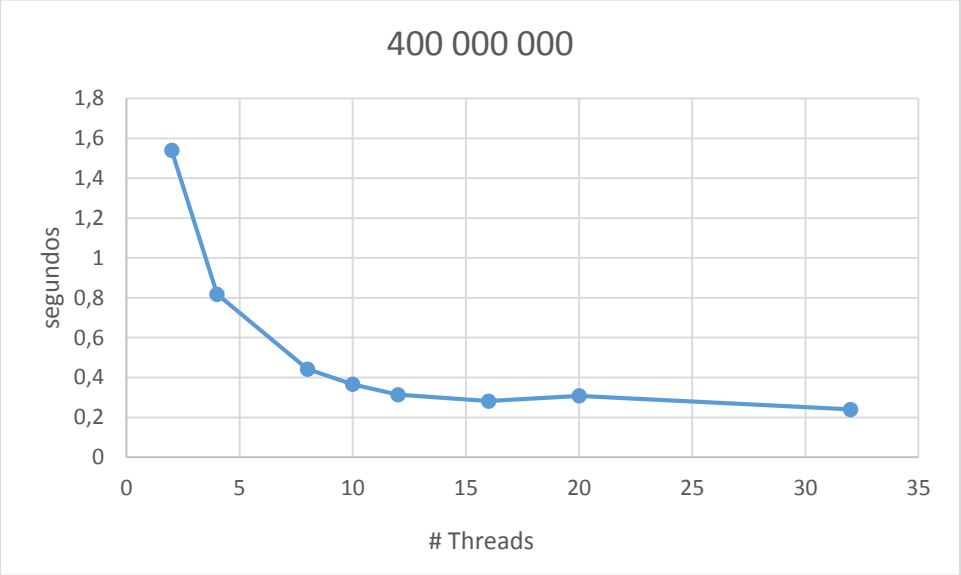
		Tests						
		# 1	**# 2**	**# 3**	**# 4**	**# 5**	**Average**	**Speedup**
Threads	**2**	0,78	0,77	0,8	0,76	0,79	0,78	1
	4	0,4	0,4	0,41	0,41	0,42	0,408	1,911765
	8	0,23	0,23	0,22	0,23	0,24	0,23	3,391304
	10	0,18	0,19	0,2	0,19	0,19	0,19	4,105263
	12	0,17	0,15	0,17	0,16	0,15	0,16	4,875
	16	0,18	0,12	0,13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

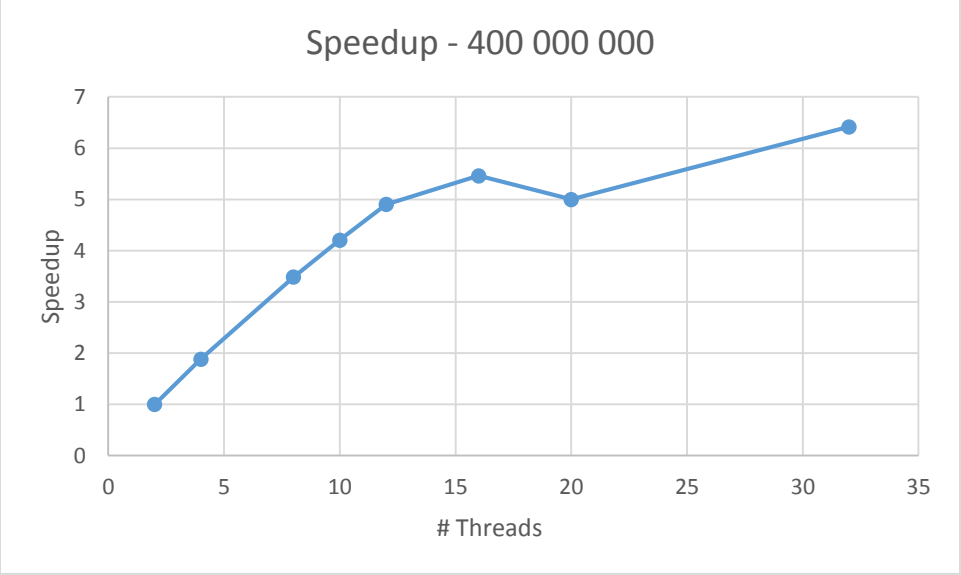




100 Milhões de Elementos

		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.