SOMA MÁXIMA DE UMA SUBSEQUÊNCIA DE TAMANHO K USANDO OPENMP

Daniela Amaral da Rosa <daniela.rosa@edu.pucrs.br>
Vinicius Braz Lima <vinius.braz@edu.pucrs.br>
Roland Teodorowitsch[1] <roland.teodorowitsch@pucrs.br> -Orientador

Pontifícia Universidade Católica do Rio Grande do Sul – Faculdade de Informática –
Bairro Partenon – CEP 90619-900 – Porto Alegre – RS

6 de maio de 2021

RESUMO

Este artigo descreve os resultados obtidos na execução do primeiro trabalho da cadeira de Programação Paralela do primeiro semestre de 2021 do curso de Engenharia de Software da Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS). Ao longo deste artigo serão apresentadas as estratégias de implementação utilizadas com uso da ferramenta OpenMP os resultados obtidos na paralelização.

Palavras-chave: Engenharia de Software; Programação Paralela.

ABSTRACT

Title: *"Maximum sum subsequence of length k using OpenMP"*

This paper describes the results obtained in the execution of the first work of the chair of Parallel Programming in the first semester of 2021 of the Software Engineering course at the Pontifical Catholic University of Rio Grande do Sul (PUCRS). Throughout this article, the implementation strategies used with the use of the OpenMP tool will be presented, the results obtained in the parallelization.

Key-words: *Software Engineer; Parallel Programming*

1 INTRODUÇÃO

O trabalho proposto tinha por objetivo implementar uma versão paralela, usando a ferramenta OpenMP, de um programa para determinar a soma máxima de uma subsequência de tamanho k de um vetor de internos, e realizar testes no cluster grad do Laboratório de Alto Desempenho da PUCRS. A implementação da solução baseia-se em código-fonte em linguagem C disponibilizado pelo professor.

2 DESENVOLVIMENTO

2.1 Ambiente de teste

Os programas foram executados no cluster grad do Laboratório de Alto Desempenho da Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) utilizando Batchjobs para agendamentos das execuções nas máquinas. As configurações de CPU das máquinas dos nodos do cluster foram extraídas no dia 06/05/2021 do nodo grad03 conforme Figura 1 e Figura 2.

1 Professor das disciplinas de Introdução à Ciência da Computação e Programação Distribuída do curso de Ciência da Computação, e da disciplina de Sistemas Distribuídos do curso de Sistemas de Informação, da Faculdade de Informática da PUCRS.

```
pp12710@grad03:/proc$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                16
On-line CPU(s) list:   0-15
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 26
Stepping:              5
CPU MHz:               2260.962
BogoMIPS:              4521.85
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15
```

Figura 1 – Resultado do comando lscpu na máquina grad03

```
model name      : Intel(R) Xeon(R) CPU           E5520  @ 2.27GHz
stepping        : 5
microcode       : 0x11
cpu MHz         : 2260.962
cache size      : 8192 KB
```

Figura 2 – Resultado do comando more cpuinfo na máquina grad03

2.2 Testes realizados

Primeiramente, foi executada a versão sequencial do algoritmo disponibilizada pelo professor junto ao enunciado do trabalho, os tempos de execução gerados na saída de erro padrão (stderr) encontram-se na Figura 3.

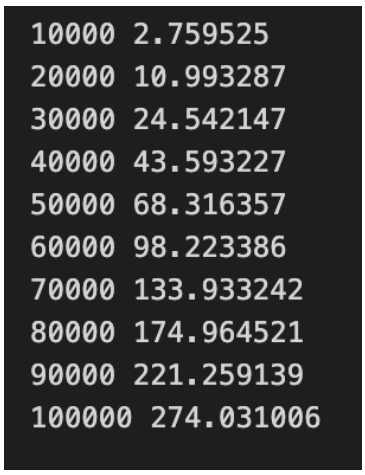

Figura 3 – Saída de erro padrão do algoritmo sequencial disponibilizado

Para a paralelização do programa, foi feita uma alteração nos laços iterativos do algoritmo para remover dependência de dados existente, ficando o código-fonte organizado conforme a Figura 4. Para podermos comparar os tempos de execução com as diretivas do OpenMP, o programa da Figura 4 foi executado duas vezes, de forma sequencial, e obteve uma piora nos tempos de execução, conforme mostra Figura 5.

```
for (l = 1; l <= k - 1; l++) {
    for (i = 1; i < n; i++) {
        for (j = 0; j < i; j++) {
            if (arr[j] < arr[i] && dp[j][l] != -1) {
                dp[i][l + 1] = MAX(dp[i][l + 1],dp[j][l] + arr[i]);
            }
        }
    }
}
```

Figura 4 – Trecho do algoritmo após remoção de dependência dos laços

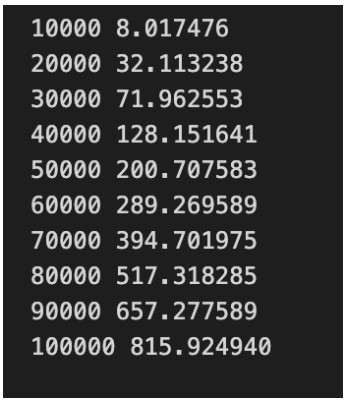

Figura 5 – Tempos de execução do algoritmo da Figura 4

Após as execuções sequenciais, começamos a realizar testes incluindo as diretivas do OpenMP para paralelizar as execuções dos laços iterativos do programa. O primeiro teste de paralelização foi feito utilizando a diretiva de paralelização de laços, e declarando a variável do laço mais interno como privada, garantindo que cada thread tivesse sua cópia da variável, a fim de manter os resultados corretos, conforme Figura 6.

```
for (l = 1; l <= k - 1; l++) {
    #pragma omp parallel for private(j)
    for (i = 1; i < n; i++) {
        for (j = 0; j < i; j++) {
            if (arr[j] < arr[i] && dp[j][l] != -1) {
                dp[i][l + 1] = MAX(dp[i][l + 1],dp[j][l] + arr[i]);
            }
        }
    }
}
```

Figura 6 – Código-fonte do primeiro teste de execução paralela

Para o código-fonte da Figura 5, fizemos execuções com 2, 4, 8 e 16 threads executando de maneira paralela, e obtivemos resultados bastante expressivos de melhora de desempenho, principalmente com 16 threads, conforme saída apresentada na Figura 7.

```
10000 1.117901
20000 3.949840
30000 8.701069
40000 15.569609
50000 24.437488
60000 36.732255
70000 62.174669
80000 83.085089
90000 101.500635
100000 128.646595
```

Figura 7 – Tempos de execução com 16 threads do código da Figura 6

Analisando o funcionamento do algoritmo dado do problema, e buscando simular as diretivas apresentadas nas aulas da cadeira, vimos uma oportunidade de testar o uso de balanceamento de carga com a cláusula schedule. Pois neste algoritmo, cada iteração pode levar um tempo diferente de execução. Realizamos testes utilizando diferentes valores para o chunk, e observamos que quanto menor o tamanho, melhor o desempenho. Não observamos diferenças no tempo de execução para mesmos valores de chunk utilizando escalonamento dinâmico e auto-escalonamento guiado.

Observando outros trechos do código, vimos oportunidade de testar o uso de diretiva de escalonamento condicional de laços e decidimos por aplicar condição para valor de entrada n maior do que 10000, pois percebemos que a paralelização no trecho somente surtiria efeito na redução do tempo para um número maior de execuções, conforme Figura 8. Com o uso desta diretiva, percebemos uma pequena melhora nos tempos de execução, por volta de 2 segundos em média.

```c
int maximum_sum_subsequence(int *arr, int n, int k) {
    int i, j, l, ans = -1;
    int **dp;

    dp = (int **)malloc(n * sizeof(int *));

    #pragma omp parallel for if(n > 10000)
    for (i=0; i<n; i++)
        dp[i] = (int *)malloc((k+1) * sizeof(int));


    for (i = 0; i < n; i++) {
        // dp[i][0] = -1; // NÃO UTILIZADO
        dp[i][1] = arr[i];
        for (j = 2; j <= k; j++)
            dp[i][j] = -1;
    }
```

Figura 8 – Código-fonte com escalonamento condicional de laço

2.3 Análise dos resultados

Para a análise final dos tempos de execução e cálculo de Speed-up e eficiência, executamos o algoritmo cinco vezes para cada configuração de número de threads, e utilizamos a configuração de escalonamento dinâmico. Os melhores tempos encontrados estão na Tabela 1. O tempo de execução sequencial considera a execução de forma sequencial do algoritmo com a implementação da alteração nos laços iterativos para possibilitar a paralelização. O Speed Up, calculado em cima dos tempos de execução da Tabela 1 encontra-se na Tabela 2, e a Eficiência encontra-se na Tabela 3.

Tabela 1 – Tempos de execução em segundos

n\Threads	Sequencial	2 Threads	4 Threads	8 Threads	16 Threads
10000	8,017476	5,033823	1,698953	0,891738	0,785846
20000	32,113238	16,394526	6,698196	3,505870	3,116857
30000	71,962553	36,719258	14,989457	7,807179	6,979365
40000	128,151641	64,970418	26,525455	13,802713	12,332011
50000	200,707583	101,946821	41,512593	21,564836	19,337442
60000	289,269589	146,628612	59,630157	30,950692	27,702132
70000	394,701975	199,873814	81,262923	42,183286	37,789660
80000	517,318285	262,684466	106,655435	55,108450	49,243489
90000	657,277589	333,116155	135,074027	69,983012	62,409254
100000	815,924940	414,932287	167,719192	86,848226	77,082547

Tabela 2 – Speed Up

n\Threads	2 Threads	4 Threads	8 Threads	16 Threads
10000	1,592721	4,719069	8,990843	10,202350
20000	1,958778	4,794311	9,159848	10,303084
30000	1,959804	4,800878	9,217485	10,310759
40000	1,972461	4,831270	9,284526	10,391788
50000	1,968748	4,834860	9,307169	10,379221
60000	1,972805	4,851062	9,346143	10,442142
70000	1,974756	4,857098	9,356833	10,444708
80000	1,969352	4,850370	9,387277	10,505313
90000	1,973118	4,866055	9,391959	10,531733
100000	1,966405	4,864828	9,394837	10,585080

Tabela 3 – Eficiência

n\Threads	2 Threads	4 Threads	8 Threads	16 Threads
10000	0,796361	1,179767	1,123855	0,637647
20000	0,979389	1,198578	1,144981	0,643943
30000	0,979902	1,200219	1,152186	0,644422
40000	0,986231	1,207818	1,160566	0,649487
50000	0,984374	1,208715	1,163396	0,648701
60000	0,986402	1,212766	1,168268	0,652634
70000	0,987378	1,214274	1,169604	0,652794
80000	0,984676	1,212592	1,173410	0,656582
90000	0,986559	1,216514	1,173995	0,658233
100000	0,983203	1,216207	1,174355	0,661568

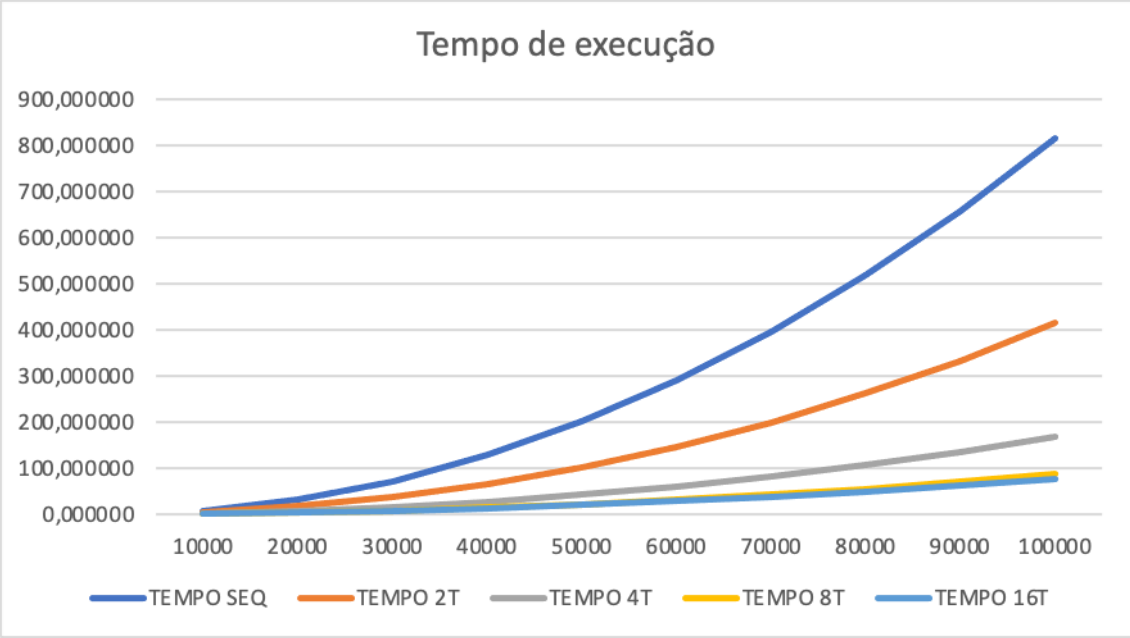

Figura 9 – Gráfico de tempo de execução de quantidade de threads em função da entrada n

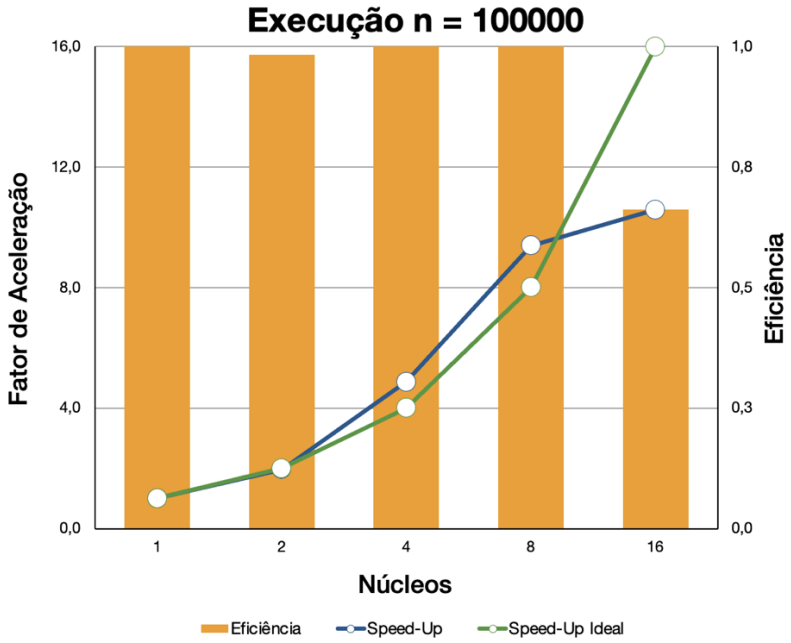

Figura 10 – Gráfico de Speed Up e Eficiência para valor de n igual a 100000

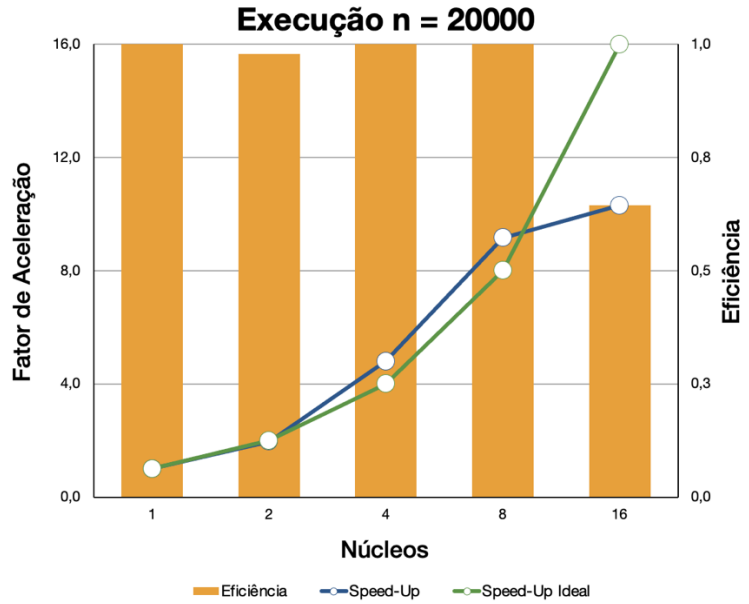

Figura 11 – Gráfico de Speed Up e Eficiência para valor de n igual a 20000

Observando os gráficos das Figuras 10 e 11, podemos notar que a inclinação das curvas de Speed Up e Eficiência é a mesma para as diferentes entradas de n do programa.

Ao compararmos as execuções paralelas com a execução sequencial do algoritmo original, disponibilizada pelo professor, podemos notar uma configuração diferente de gráfico de Speed Up e Eficiência conforme a Figura 12. Vemos que apesar do tempo de execução reduzir bastante da execução sequencial para a execução com 16 threads, o speed up está muito abaixo do ideal, e a eficiência tende a diminuir quanto mais recursos utilizados.

Núcleos	Tempo de Execução (s)	Speed-Up	Speed-Up Ideal	Eficiência
1	274,0310060	1,0	1	1,0
2	414,932287	0,7	2	0,3
4	167,719192	1,6	4	0,4
8	86,848226	3,2	8	0,4
16	77,082547	3,6	16	0,2

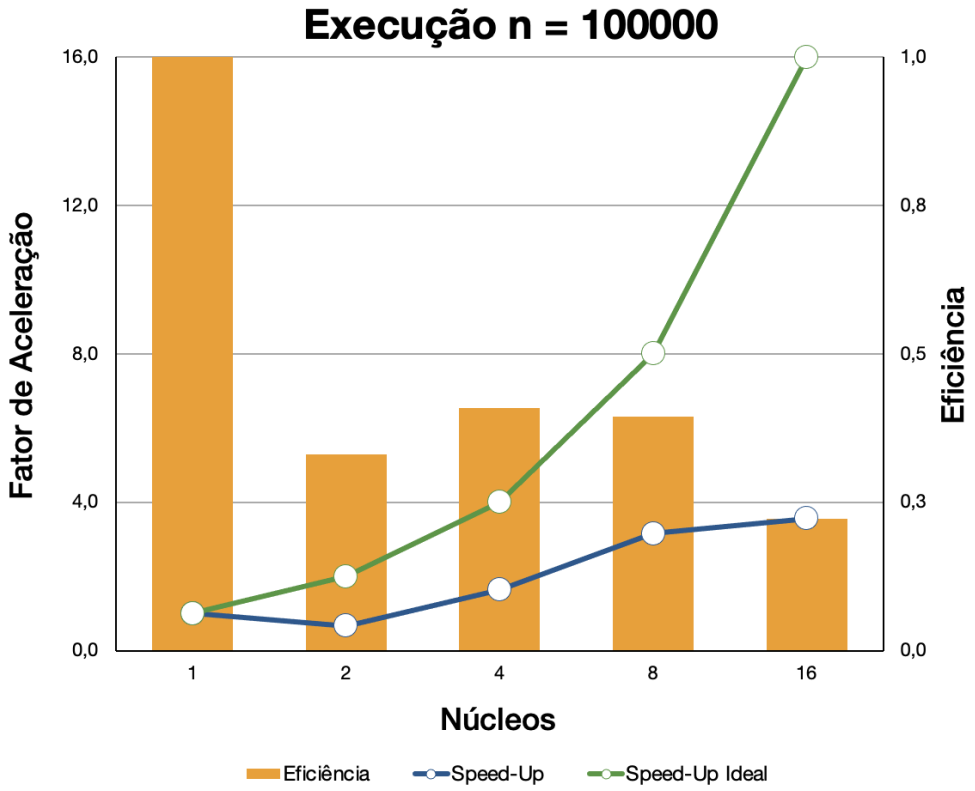

Figura 12 – Gráfico de Speed Up e Eficiência para valor de n igual a 10000 e versão sequencial original

Nas Tabelas 4 e 5 estão os cálculos de Speed Up e Eficiência das execuções paralelas comparadas com a execução sequencial do algoritmo original, onde percebemos que devido a alteração na ordem dos laços iterativos, a execução somente ficou mais rápida com um maior número de threads sendo executadas de forma paralela.

Tabela 4 – Speed Up comparado com sequencial original

n\Threads	2 Threads	4 Threads	8 Threads	16 Threads
10000	0,548197	1,624250	3,094547	3,511534
20000	0,670546	1,641231	3,135680	3,527042
30000	0,668373	1,637294	3,143536	3,516387
40000	0,670970	1,643449	3,158309	3,534965
50000	0,670118	1,645678	3,167952	3,532854
60000	0,669879	1,647210	3,173544	3,545698
70000	0,670089	1,648147	3,175031	3,544177
80000	0,666063	1,640465	3,174913	3,553049
90000	0,664210	1,638058	3,161612	3,545294
100000	0,660423	1,633868	3,155286	3,555033

Tabela 5 – Eficiência comparada com sequencial original

n\Threads	2 Threads	4 Threads	8 Threads	16 Threads
10000	0,274098	0,406063	0,386818	0,219471
20000	0,335273	0,410308	0,391960	0,220440
30000	0,334186	0,409323	0,392942	0,219774
40000	0,335485	0,410862	0,394789	0,220935
50000	0,335059	0,411419	0,395994	0,220803
60000	0,334939	0,411802	0,396693	0,221606
70000	0,335044	0,412037	0,396879	0,221511
80000	0,333032	0,410116	0,396864	0,222066
90000	0,332105	0,409515	0,395202	0,221581
100000	0,330212	0,408467	0,394411	0,222190

3 CONCLUSÃO

Durante a realização deste trabalho tivemos a oportunidade de simular o uso de técnicas de paralelização de aplicações utilizando diretivas de compilação da ferramenta OpenMP para a linguagem de programação C. Esta ferramenta torna extremamente mais fácil esta tarefa, quando comparado com a paralelização explícita na escrita dos algoritmos. Pudemos concluir na execução dos testes deste trabalho, que a paralelização de uma aplicação pode trazer grandes ganhos de desempenho, ao utilizar de maneira completa e eficiente os recursos disponibilizados pela CPU das máquinas, segmentando a execução dos algoritmos em diferentes threads a serem executadas de maneira paralela.

REFERÊNCIAS

Slides e vídeos disponibilizados pelo professor.