DCC125 - Programação Paralela
Universidade Federal de Juiz de Fora
Novembro 2020

Prova 2

Diego Paiva e Silva (201565516C)

Obs: Em todas as tabelas, a coluna t denota o tempo médio, em segundos, coletado em 10 execuções, e σ o respectivo desvio padrão. As demais variáveis são como segue: (S) *speedup*; (E) eficiência; (n) tamanho do problema de entrada e (p) quantidade de processos/*threads* utilizados(as). Todas as execuções foram feitas em máquinas **Intel** do *cluster*.

1. (10 pontos) Use uma das paralelizações (escolha a mais eficiente) disponíveis para a integração via trapézio via OpenMP. Rode e compare o desempenho do código OpenMP com a sua paralelização usando MPI.

   Resposta: O *benchmark* das implementações do algoritmo em OpenMP e MPI são reportados nas Tabelas 1 e 2, respectivamente.

n ($\times 10^9$)	**p = 1**				**p = 2**				**p = 4**				**p = 8**			
	t	σ	S	E	t	σ	S	E	t	σ	S	E	t	σ	S	E
2	29.72	0.00	1.0	1.0	14.87	0.00	**2.00**	1.00	7.44	0.00	**4.00**	1.00	3.97	0.01	7.49	0.94
4	59.44	0.01	1.0	1.0	29.75	0.00	**2.00**	1.00	14.88	0.00	**4.00**	1.00	7.83	0.00	7.59	0.95
6	89.17	0.00	1.0	1.0	44.62	0.00	**2.00**	1.00	22.32	0.00	**4.00**	1.00	11.69	0.00	7.63	0.95
8	118.89	0.00	1.0	1.0	59.50	0.00	**2.00**	1.00	29.76	0.00	**4.00**	1.00	15.56	0.00	7.64	0.96
10	148.61	0.00	1.0	1.0	74.37	0.00	**2.00**	1.00	38.40	3.48	3.87	0.97	19.42	0.00	7.65	0.96

Tabela 1: OpenMP: resultados para cada par (n, p).

n ($\times 10^9$)	**p = 1**				**p = 2**				**p = 4**				**p = 8**			
	t	σ	S	E	t	σ	S	E	t	σ	S	E	t	σ	S	E
2	37.02	0.02	1.0	1.0	18.67	0.01	1.98	0.99	9.60	0.04	3.86	0.96	4.96	0.01	7.46	0.93
4	73.82	0.00	1.0	1.0	37.09	0.01	1.99	1.00	19.01	0.06	3.88	0.97	9.82	0.00	7.52	0.94
6	110.67	0.08	1.0	1.0	55.52	0.02	1.99	1.00	28.44	0.09	3.89	0.97	14.67	0.01	7.54	0.94
8	147.44	0.01	1.0	1.0	73.94	0.03	1.99	1.00	37.81	0.12	3.90	0.97	19.53	0.00	7.55	0.94
10	184.25	0.00	1.0	1.0	92.38	0.02	1.99	1.00	47.28	0.15	3.90	0.97	24.39	0.01	7.55	0.94

Tabela 2: MPI: resultados para cada par (n, p).

2. (20 pontos) Write an OpenMP program that implements multiplication of a vector by a scalar and dot product. The inputs are two vectors and a scalar. The results are calculated and collected onto thread 0, which prints them. You can assume that n, the order of the vectors, is evenly divisible by *comm sz*.

   Ou seja, as entradas são a (escalar), $\overrightarrow{v_1}$ e $\overrightarrow{v_2}$ (vetores de tamanho n). A saída é um escalar: $y = a\overrightarrow{v_1} \cdot \overrightarrow{v_2}$, ou seja, o produto escalar (ou produto interno) de $a\overrightarrow{v_1}$ com $\overrightarrow{v_2}$.

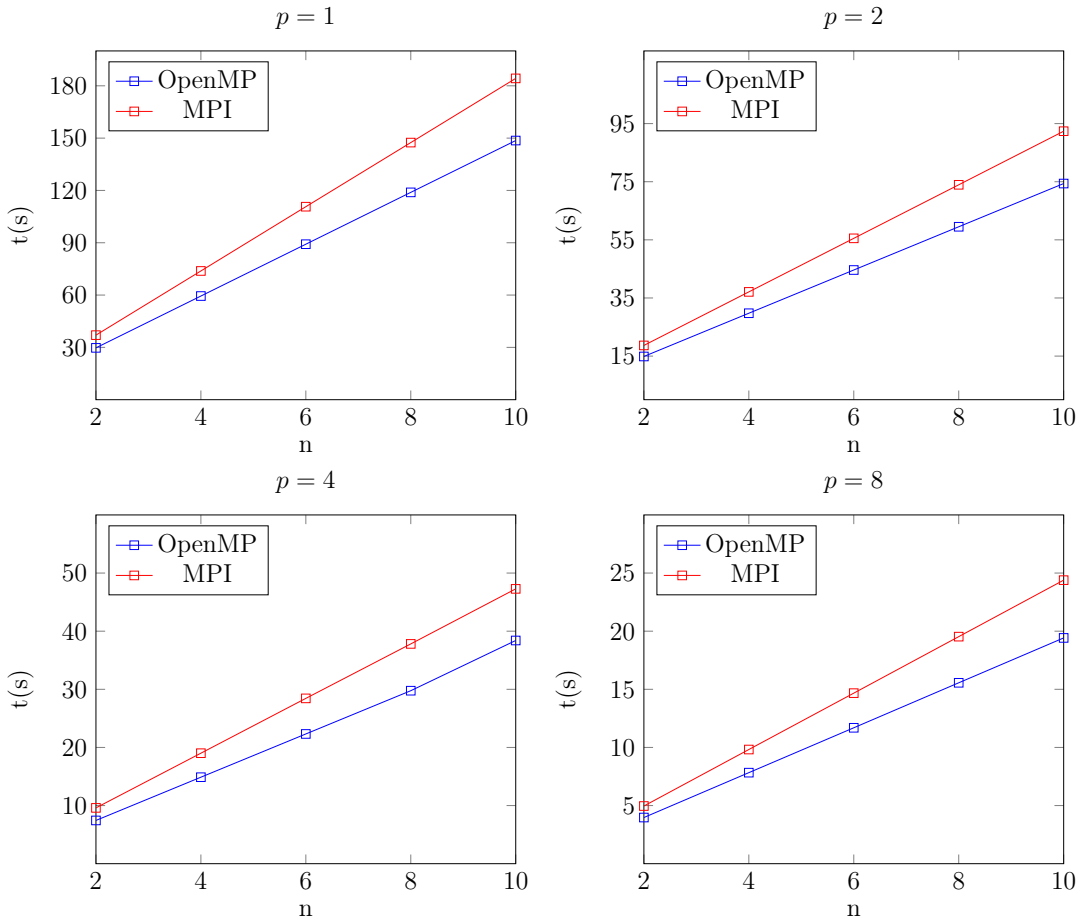

Figura 1: Comparação entre os tempos demandados pelas implementações em OpenMP e MPI da regra do trapézio.

Para aumentar o tempo de execução de seu código faça esse produto escalar K vezes (um *loop* para realizar a conta paralela K vezes, onde K pode ser 10, 100, 1000. Qualquer numero que aumente consideravelmente o tempo de execução do código sequencial.

Rode com diferentes tamanhos de n e número de processos p. Monte uma tabela com os *speedups* e eficiências. Compare o desempenho do código OpenMP com a sua paralelização usando MPI.

Resposta: O *benchmark* das implementações do algoritmo em OpenMP e MPI são reportados nas Tabelas 3 e 4, respectivamente.

n ($\times 10^7$)	**p = 1**				**p = 2**				**p = 4**				**p = 8**			
	t	σ	S	E	t	σ	S	E	t	σ	S	E	t	σ	S	E
2	10.16	0.09	1.0	1.0	5.17	0.00	1,97	0.98	2.72	0.01	3.74	0.93	1.31	0.00	7.76	0.97
4	20.05	0.00	1.0	1.0	10.14	0.00	1.98	0.99	5.26	0.01	3.81	0.95	2.63	0.00	7.62	0.95
6	29.97	0.02	1.0	1.0	15.14	0.07	1.98	0.99	7.81	0.01	3.84	0.96	3.94	0.00	7.61	0.95
8	39.90	0.03	1.0	1.0	20.26	0.32	1.97	0.98	10.35	0.02	3.86	0.96	5.81	1.74	6.87	0.86
10	49.90	0.00	1.0	1.0	25.12	0.09	1.99	0.99	12.89	0.04	3.87	0.97	6.57	0.00	7.60	0.95

Tabela 3: OpenMP: resultados para cada par (n, p).

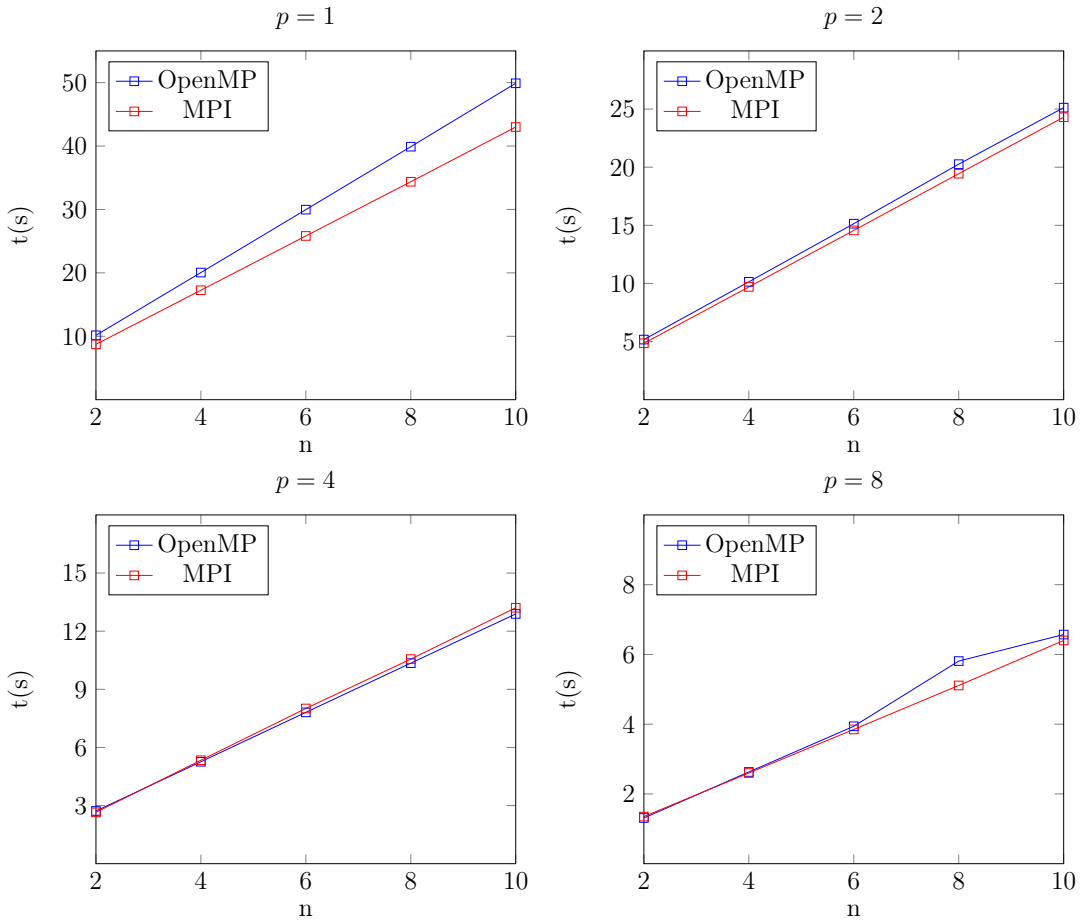

Figura 2: Comparação entre os tempos demandados pelas implementações em OpenMP e MPI do produto escalar e multiplicação por escalar.

n ($\times 10^7$)	**p = 1**				**p = 2**				**p = 4**				**p = 8**			
	t	σ	S	E	t	σ	S	E	t	σ	S	E	t	σ	S	E
2	8.73	0.02	1.0	1.0	4.85	0.02	1.80	0.90	2.65	0.04	3.29	0.82	1.35	0.01	6.47	0.81
4	17.26	0.04	1.0	1.0	9.70	0.04	1.78	0.89	5.34	0.08	3.23	0.81	2.60	0.02	6.64	0.83
6	25.82	0.06	1.0	1.0	14.56	0.06	1.77	0.87	8.01	0.10	3.22	0.81	3.85	0.03	6.71	0.84
8	34.37	0.00	1.0	1.0	19.44	0.03	1.77	0.88	10.56	0.14	3.25	0.81	5.11	0.04	6.73	0.84
10	43.00	0.06	1.0	1.0	24.30	0.03	1.77	0.88	13.21	0.17	3.26	0.81	6.40	0.05	6.72	0.84

Tabela 4: MPI: resultados para cada par (n, p).

3. (20 pontos) Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are 2 feet in length. Suppose also that there's a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and it's area is π square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation number in circle/total number of tosses $= \pi/4$, since the ratio of the area of the circle to the area of the square is $\pi/4$. We can use this formula to estimate the value of π with a random number generator:

```
for (toss = 0; toss < number of tosses; toss++) {
    x = random double between -1 and 1;
```

```
y = random double between -1 and 1;
distance squared = x*x + y*y;

if (distance squared <= 1)
    number in circle++;
}

pi estimate = 4*number in circle/((double) number of tosses);
```

This is called a "Monte Carlo" method, since it uses randomness (the dart tosses). Write an OpenMP program that uses a Monte Carlo method to estimate π. Thread 0 should read in the total number of tosses. Use `MPI_Reduce` to find the global sum of the local variable number in circle, and have thread 0 print the result. You may want to use `long long int`s for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of π.

Rode com diferentes valores de *number of tosses* e número de processos p. Monte uma tabela com os *speedups* e eficiências. Compare o desempenho do código OpenMP com a sua paralelização usando MPI.

Resposta: O *benchmark* das implementações do algoritmo em OpenMP e MPI são reportados nas Tabelas 5 e 6, respectivamente.

n ($\times 10^9$)	**p = 1**				**p = 2**				**p = 4**				**p = 8**			
	t	σ	S	E	t	σ	S	E	t	σ	S	E	t	σ	S	E
2	80.07	0.18	1.0	1.0	35.05	0.32	**2.28**	1.14	17.64	0.04	**4.54**	1.13	9.23	0.02	**8.67**	1.08
4	160.24	0.12	1.0	1.0	70.06	0.77	**2.29**	1.14	35.07	0.05	**4.57**	1.14	18.34	0.03	**8.74**	1.09
6	240.44	0.28	1.0	1.0	104.32	0.31	**2.30**	1.15	52.49	0.12	**4.58**	1.15	27.44	0.01	**8.76**	1.10
8	320.44	0.63	1.0	1.0	139.12	0.85	**2.30**	1.15	70.63	2.30	**4.54**	1.13	36.57	0.03	**8.76**	1.10
10	400.68	0.44	1.0	1.0	174.60	0.90	**2.29**	1.15	87.42	0.16	**4.58**	1.15	45.91	0.10	**8.73**	1.09

Tabela 5: OpenMP: resultados para cada par (n, p).

n ($\times 10^9$)	**p = 1**				**p = 2**				**p = 4**				**p = 8**			
	t	σ	S	E	t	σ	S	E	t	σ	S	E	t	σ	S	E
2	124.93	5.15	1.0	1.0	64.21	4.87	1.95	0.97	33.40	4.01	3.74	0.94	17.87	1.79	6.99	0.87
4	248.60	9.19	1.0	1.0	123.20	1.68	**2.02**	1.01	63.16	3.10	3.94	0.98	32.98	2.50	7.53	0.94
6	368.69	6.16	1.0	1.0	185.85	6.69	1.98	0.99	101.96	14.36	3.62	0.90	50.85	6.23	7.25	0.91
8	498.05	27.61	1.0	1.0	251.95	14.57	1.98	0.99	125.75	5.80	3.96	0.99	70.08	7.74	7.11	0.89
10	644.92	75.29	1.0	1.0	326.10	43.09	1.98	0.99	166.04	20.50	3.88	0.97	86.24	9.65	7.48	0.93

Tabela 6: MPI: resultados para cada par (n, p).

Conclusões

Observando os resultados gerais, podemos concluir que o OpenMP tende a fornecer melhor desempenho para problemas relativamente grandes, ou seja, aqueles que demandam um tempo de computação razoável na execução serial. Este comportamento pôde ser observado nas questões 1 e 3, nas quais o OpenMP se saiu melhor justamente devido ao fato dos valores de n exigirem uma computação mais longa. Por outro lado, na questão

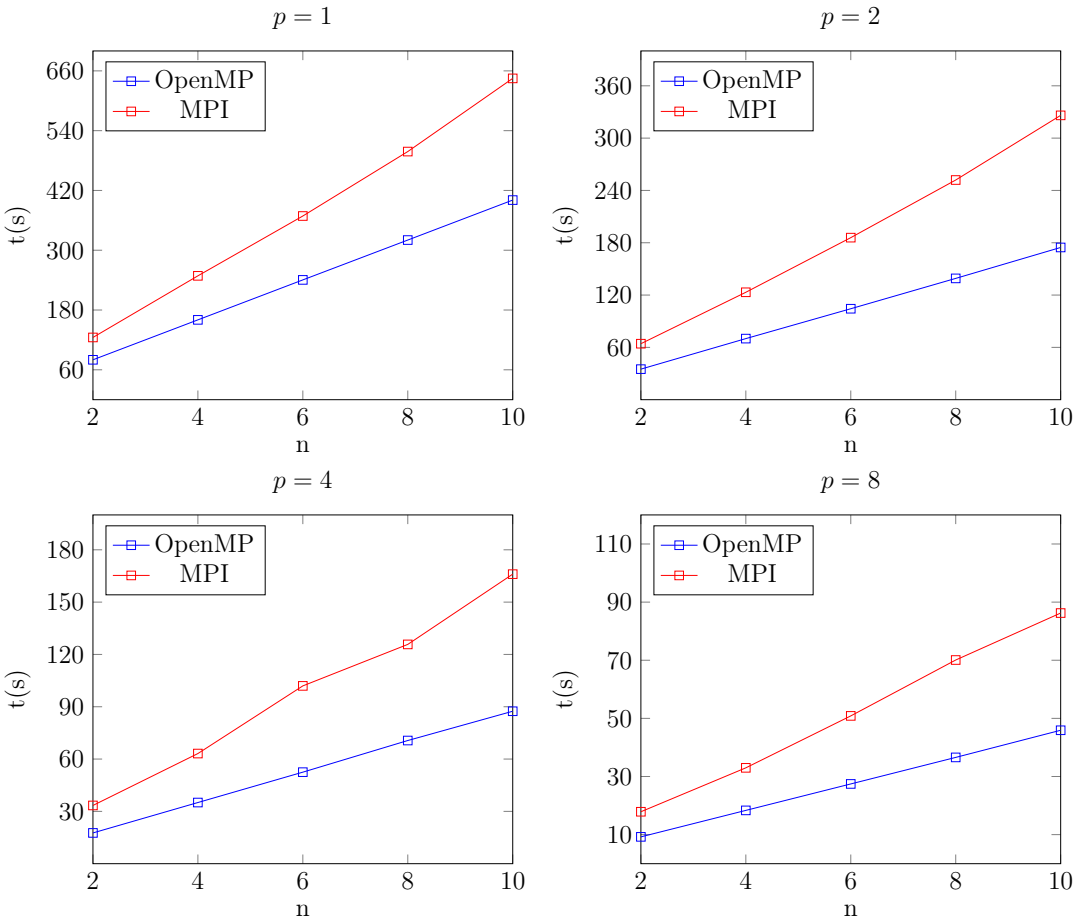

Figura 3: Comparação entre os tempos demandados pelas implementações em OpenMP e MPI do método de Monte Carlo para determinação do valor de π.

2, a diferença entre as duas bibliotecas não foi tão significativa, o que pode ser justificado pelo tempo relativamente baixo exigido para a computação de cada valor de n. O fato de o MPI ser baseado em troca de mensagens, e não em memória compartilhada, justifica o desempenho inferior em relação ao OpenMP, pois tal característica provoca um *overhead* de comunicação, e isso tende a piorar quanto mais alto for o valor de p, pois intensificará a quantidade de mensagens trocadas entre os processos.