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

Prova 1

Diego Paiva e Silva (201565516C)

Pra todas as medições de tempos, sempre execute os códigos ao menos 5 vezes e apresente a média de desvio padrão dos tempos de execução.

1. (10 pontos) Modify the trapezoidal rule so that it will correctly estimate the integral even if *comm_sz* doesn't evenly divide *n*. (You can still assume that n $\geq$ *comm_sz*.)

   Rode o código e verifique que independente do numero de *cores*, o mesmo resultado final para a integral numérica está sendo obtido. Ou seja, verifique que o código está funcionando (corretude).

   Resposta:

```
1  h = (b-a)/n;          /* h is the same for all processes */
2  local_n = n/comm_sz; /* So is the number of trapezoids  */
3
4  /* Se a conta não for exata, reparta o resto p/ demais processos */
5  if (n % comm_sz > my_rank) {
6    local_n += 1;
7    local_a = a + my_rank * local_n * h;
8  }
9  else {
10   /* Ajusta o valor de local_a para aqueles que não recebem resto */
11   local_a = a + (my_rank * local_n + n % comm_sz) * h;
12 }
13
14 local_b = local_a + local_n * h;
```

   Exemplos de execução para prova de corretude:

```
$ mpiexec -n 1 ./mpi_trap3 100
With n = 100 trapezoids, our estimate of the integral
from 0.000000 to 3.000000 = 9.000449999999995e+00

$ mpiexec -n 2 ./mpi_trap3 100
With n = 100 trapezoids, our estimate of the integral
from 0.000000 to 3.000000 = 9.000449999999997e+00

$ mpiexec -n 3 ./mpi_trap3 100
With n = 100 trapezoids, our estimate of the integral
from 0.000000 to 3.000000 = 9.000449999999999e+00
```

2. (20 pontos) Write an MPI program that implements multiplication of a vector by a scalar and dot product. The user should enter two vectors and a scalar, all of which are read in by process 0 and distributed among the processes. The results are calculated and collected onto process 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 $\overrightarrow{v_2}$.

   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. Responda se o código é fortemente escalável ou fracamente escalável.

   Resposta:

   Os resultados da implementação do algoritmo em linguagem C (média de 5 execuções para cada par (n, p)) são reportados na Tabela 1.

n ($\times 10^6$)	**p = 1**				**p = 2**				**p = 4**			
	T (s)	SD	S	E	T (s)	SD	S	E	T (s)	SD	S	E
100	0,49	0,008	1,0	1,0	0,44	0,004	1,11	0,56	0.46	0,005	1,06	0,26
200	0,97	0,0136	1,0	1,0	0,85	0,00633	1,14	0,57	0,99	0,175	**0,97**	0,24
300	1,63	0,20	1,0	1,0	1,27	0,00748	1,28	0,64	1,34	0,0258	1,22	0,30
400	2,10	0,261	1,0	1,0	1,69	0,016	1,24	0,62	1,84	0,0379	1,14	0,29
500	4,25	0,843	1,0	1,0	2,79	0,0788	1,52	0,76	3,83	0,185	1,11	0,28

Tabela 1: Tempo (T), Desvio Padrão (SD), *Speedup* (S) e Eficiência (E) para cada par (n, p) em i7-8565U @ 1.80GHz.

   A partir dos resultados, percebe-se que o algoritmo não é fortemente escalável, pois não obtém *speedup* linear para nenhum $p > 1$. Curiosamente, talvez em razão da maneira como a implementação foi feita, os *speedups* foram praticamente insignificantes e houve um único caso em que o algoritmo paralelo teve desempenho inferior ao sequêncial, em $(n = 200, p = 4)$. Isto pode ser justificado, possivelmente, devido ao *overhead* de comunicação, pois com $p = 4$ os *speedups* foram piores que aqueles registrados com $p = 2$. Ademais, como a eficiência mostrou uma tendência de queda acentuada a medida que aumentaram-se os valores de n e p (chegando, inclusive, a patamares menores que 30%), o código também não demonstrou ser fracamente escalável.

3. (20 pontos) Find the speedups and efficiencies of the parallel odd-even sort. Does the program obtain linear speedups? Is it scalable? Is it strongly scalable? Is it weakly scalable?

Resposta:

Os resultados da implementação do algoritmo em linguagem C (média de 5 execuções para cada par (n, p)) são reportados na Tabela 2.

n	**p = 1**				**p = 2**				**p = 4**			
	T (s)	SD	S	E	T (s)	SD	S	E	T (s)	SD	S	E
2^{24}	1,96	0,0049	1,0	1,0	1,34	0,0194	1,46	0,73	0,92	0,0253	2,13	0,53
2^{25}	4,47	0,481	1,0	1,0	2,77	0,0117	1,61	0,81	1,91	0,00748	2.34	0,58
2^{26}	8,96	0,131	1,0	1,0	5,67	0,00748	1,58	0,79	3,90	0,0136	2,30	0,57
2^{27}	18,24	0,05	1,0	1,0	11,62	0,0224	1,57	0,78	7,94	0,0232	2,30	0,57
2^{28}	37,43	0,105	1,0	1,0	23,88	0,0508	1,57	0,78	16,28	0,0614	2,30	0,57

Tabela 2: Tempo (T), Desvio Padrão (SD), *Speedup* (S) e Eficiência (E) para cada par (n, p) em i7-8565U @ 1.80GHz.

A partir dos resultados reportados, é possível observar que o algoritmo é incapaz de obter *speedups* lineares pois em nenhuma das execuções para $p > 1$ o valor de S se aproximou de p, logo ele não é fortemente escalável. Além disso, o *parallel odd-even sort* também demonstrou não ser fracamente escalável, pois a medida que n e p aumentam, a eficiência E tende a decrescer significativamente. Por motivo de limitação de *hardware*, não foi possível observar este comportamento para outros valores de p, porém certamente a eficiência manteria uma tendência de queda.

4. (10 pontos) Serial odd-even transposition sort of an n-element list can sort the list in considerably fewer than n phases. As an extreme example, if the input list is already sorted, the algorithm requires 0 phases.

   (a) Write a serial `Is_sorted` function that determines whether a list is sorted.

   Resposta:

```c
int Is_sorted(int a[], int n) {
   for (int i = 1; i < n; i++)
      if (a[i - 1] > a[i])
         return 0;

   return 1;
}
```

   (b) Modify the serial odd-even transposition sort program so that it checks whether the list is sorted after each phase.

   Resposta:

```c
void Odd_even_sort(int a[], int n) {
   for (int phase = 0; phase < n; phase++) {
      if (phase % 2 == 0) { /* Even phase */
         for (int i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) {
               int temp = a[i];
               a[i] = a[i-1];
```

```
8                   a[i-1] = temp;
9               }
10          } else { /* Odd phase */
11              for (int i = 1; i < n-1; i += 2)
12                  if (a[i] > a[i+1]) {
13                      int temp = a[i];
14                      a[i] = a[i+1];
15                      a[i+1] = temp;
16                  }
17          }
18
19          if (Is_sorted(a, n) == 1)
20              break;
21      }
22  }
```

(c) If this program is tested on a random collection of n-element lists, roughly what fraction get improved performance by checking whether the list is sorted?

Resposta:

Considerando que uma lista com n elementos possui probabilidade $\frac{1}{n!}$ de estar ordenada e que $\lim_{n\to\infty} \frac{1}{n!} = 0$, então concluímos que a introdução deste passo não fornece melhoria significativa no desempenho geral do algoritmo.

5. (20 pontos) Programe uma versão paralela do código serial da questão 4. Rode com diferentes tamanhos de n e número de processos p. Monte uma tabela com os *speedups* e eficiências. Responda se o código é fortemente escalável ou fracamente escalável. Lembre-se de sempre gerar o vetor de entrada de forma aleatória.

   Resposta:

   Tentei fazer uma paralelização da função, mas que não funcionou muito corretamente. Por este motivo não fiz um *benchmark* desta questão. Contudo, o código da tentativa está em `mpi_is_sorted.c`.

6. (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++;
    }
```

```
9
10      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 MPI program that uses a Monte Carlo method to estimate π. Process 0 should read in the total number of tosses and broadcast it to the other processes. Use `MPI_Reduce` to find the global sum of the local variable number in circle, and have process 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. Responda se o código é fortemente escalável ou fracamente escalável.

Resposta:

Os resultados da implementação do algoritmo em linguagem C (média de 5 execuções para cada par ((**tosses**, p)) são reportados na Tabela 3.

tosses ($\times 10^6$)	**p = 1**				**p = 2**				**p = 4**			
	T (s)	SD	S	E	T (s)	SD	S	E	T (s)	SD	S	E
100	2,14	0,008	1,0	1,0	1,41	0,0174	1,52	0,76	0,95	0,0219	2,25	0,56
300	7,06	0,575	1,0	1,0	4,32	0,120	1,63	0,82	2,82	0,0807	2,50	0,63
500	11,65	0,216	1,0	1,0	7,17	0,191	1,62	0,81	4,72	0,122	2,47	0,62
700	16,28	0,275	1,0	1,0	9,90	0,0117	1,64	0,82	6,50	0,114	2,50	0,63
900	21,07	0,614	1,0	1,0	12,74	0,0133	1,65	0,83	8,74	0,501	2,41	0,60

Tabela 3: Tempo (T), Desvio Padrão (SD), *Speedup* (S) e Eficiência (E) para cada par ((**tosses**, p)) em i7-8565U @ 1.80GHz.

Observando os valores, podemos afirmar que o algoritmo para estimar o valor de π não é fortemente escalável, pois não obtém *speedup* linear para nenhuma entrada. Por outro lado, ele também não é fracamente escalável, pois a medida que a entrada e o número de processos aumentam, a eficiência tem uma tendência acentuada de queda (basta observar as diagonais das eficiências).