Programação de Arquiteturas com Memória Compartilhada Utilizando OpenMP - Parte 2

MCZA020-13 - Programação Paralela

Emilio Francesquini
e.francesquini@ufabc.edu.br

2020.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

Disclaimer

UFABC

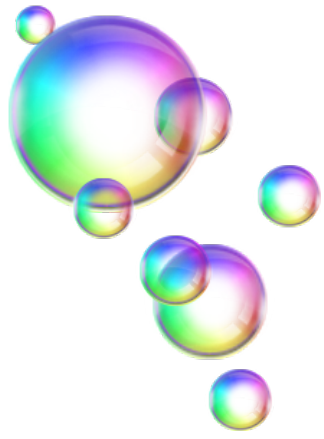
- Estes slides foram preparados para o curso de Programação Paralela na UFABC.
- Estes slides são baseados naqueles produzidos por Peter Pacheco como parte do livro *An Introduction to Parallel Programming* disponíveis em: https://www.cs.usfca.edu/~peter/ipp/
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Algumas figuras foram obtidas em: http://pngimg.com

Laços em OpenMP Parte 2 - Ordenação

A estrela mal compreendida - Bubble Sort

UFABC

```
1  void Bubble_sort(
2                   int a[] /* in/out */,
3                   int n   /* in */) {
4      int list_length, i, temp;
5
6      for (list_length = n; list_length >= 2;
       ↪  list_length--)
7          for (i = 0; i < list_length - 1; i++)
8              if (a[i] > a[i+1]) {
9                  temp = a[i];
10                 a[i] = a[i+1];
11                 a[i+1] = temp;
12             }
13 }
```



Ordenação por transposição par-ímpar

UFABC

- Funciona em uma sequência de fases
- Fases pares - comparam e trocam iniciando dos índices pares
  - `(a[0],a[1]),(a[2],a[3]),(a[4],a[5])...`
- Fases ímpares - comparam e trocam iniciando dos índices ímpares
  - `a[0],(a[1],a[2]),(a[3],a[4]),(a[5],a[6])...`

Exemplo

UFABC

- Começo: `5, 9, 4, 3`
- Fase par: compara e troca `(5, 9), (4, 3)`
  - Resultado: `5, 9, 3, 4`
- Fase ímpar: compara e troca `(9, 3)`
  - Resultado: `5, 3, 9, 4`
- Fase par: compara e troca `(5, 3), (9, 4)`
  - Resultado: `3, 5, 4, 9`
- Fase ímpar: compara e troca `(5, 4)`
  - Resultado: `3, 4, 5, 9`

Código sequencial

UFABC

```
1  void Odd_even_sort(int a[], int n) {
2      int phase, i, temp;
3      for (phase = 0; phase < n; phase++)
4          if (phase % 2 == 0) { /* Even phase */
5              for (i = 1; i < n; i += 2)
6                  if (a[i - 1] > a[i]) {
7                      temp = a[i];
8                      a[i] = a[i - 1];
9                      a[i - 1] = temp;
10                 }
11         } else { /* Odd phase */
12             for (i = 1; i < n - 1; i += 2)
13                 if (a[i] > a[i+1]) {
14                     temp = a[i];
15                     a[i] = a[i+1];
16                     a[i+1] = temp;
17                 }
18         }
19 }
```

OpenMP - Primeira Versão

UFABC

```
1  void Odd_even_sort(int a[], int n) {
2    int phase, i, temp;
3    for (phase = 0; phase < n; phase++)
4      if (phase % 2 == 0) { /* Even phase */
5  #     pragma omp parallel for num_threads(thr_count) \
6          default(none) shared(a, n) private(i, tmp)
7        for (i = 1; i < n; i += 2)
8          if (a[i - 1] > a[i]) {
9            temp = a[i]; a[i] = a[i - 1]; a[i - 1] = temp;
10         }
11     } else { /* Odd phase */
12 #     pragma omp parallel for num_threads(thr_count) \
13         default(none) shared(a, n) private(i, tmp)
14       for (i = 1; i < n - 1; i += 2)
15         if (a[i] > a[i+1]) {
16           temp = a[i]; a[i] = a[i+1]; a[i+1] = temp;
17         }
18     }
19 }
```

OpenMP - Segunda Versão

UFABC

```
1  void Odd_even_sort(int a[], int n) {
2      int phase, i, temp;
3  #   pragma omp parallel num_threads(thread_count) \
4        default(none) shared(a, n) private(i, tmp, phase)
5      for (phase = 0; phase < n; phase++)
6        if (phase % 2 == 0) { /* Even phase */
7  #       pragma omp for
8          for (i = 1; i < n; i += 2)
9            if (a[i - 1] > a[i]) {
10             temp = a[i]; a[i] = a[i - 1]; a[i - 1] =
                 ↪  temp;
11           }
12       } else { /* Odd phase */
13 #       pragma omp for
14         for (i = 1; i < n - 1; i += 2)
15           if (a[i] > a[i+1]) {
16             temp = a[i]; a[i] = a[i+1]; a[i+1] = temp;
17           }
18       }
```

Comparação de desempenho entre as duas versões

UFABC

Número de threads	1	2	3	4
Duas diretivas `parallel for`	0.770	0.453	0.358	0.305
Duas diretivas `for`	0.732	0.376	0.294	0.239

Escalonamento de laços

Escalonamento de laços

- Queremos paralelizar o seguinte laço

```
1  sum = 0.0
2  for (i = 0; i <= n; i++)
3      sum += f(i);
```

Thread	Iterações
0	0, n/t, 2n/t, ...
1	1, n/t + 1, 2n/t + 1, ...
...	...
t - 1	t - 1, n / t + t - 1, 2n / t + t - 1, ...

- Atribuição cíclica

Definindo f

UFABC

```
1  double f (int i) {
2      int j, start = i * (i + 1) /2, finish = start + i;
3      double return_val = 0.0;
4      for (j = start; j <= finish; j++) {
5          return_val += sin(j);
6      }
7      return return_val;
8  }
```

UFABC

- `f(i)` chama a função `sin`, `i` vezes
- Assuma que o tempo para executar `f(2 * i)` seja aproximadamente o dobro do tempo de chamar `f(i)`

Considerando $n = 10000$

Threads	Atribuição	Tempo	Speedup
1	-	3.67	1
2	default	2.76	1.33
2	cíclico	1.84	1.99

A cláusula `schedule`

UFABC

- Default schedule

```
1  sum = 0.0
2  #pragma omp parallel for num_threads(thread_count) \
3      reduction(+:sum)
4  for (i = 0; i <= n; i++)
5      sum += f(i);
```

- Cyclic schedule

```
1  sum = 0.0
2  #pragma omp parallel for num_threads(thread_count) \
3      reduction(+:sum) schedule(static, 1)
4  for (i = 0; i <= n; i++)
5      sum += f(i);
```

~schedule (type, chunksize)

UFABC

- `type` pode ser:
  - static - as iterações podem ser atribuídas aos threads antes do loop ser executado.
  - dynamic ou guided - as iterações são atribuídas aos threads enquanto o laço está executando.
  - auto - o compilador e/ou o ambiente de execução determinam o schedule.
  - runtime - o escalonamento é determinado durante a execução (através de uma configuração passada como variável de ambiente, por exemplo).
- O `chunksize` deve ser um inteiro positivo.

Escalonamento `static`

UFABC

- Doze iterações: 0,1,2,...11 e três threads

```
schedule(static, 1)
```

Thread Rank	Iterações
0	0, 3, 6, 9
1	1, 4, 7, 10
2	2, 5, 8, 11

Escalonamento `static`

UFABC

- Doze iterações: 0,1,2,...11 e três threads

```
schedule(static, 2)
```

Thread Rank	Iterações
0	0, 1, 6, 7
1	2, 3, 8, 9
2	4, 5, 10, 11

Escalonamento `static`

UFABC

- Doze iterações: 0,1,2,...11 e três threads

```
schedule(static, 4)
```

Thread Rank	Iterações
0	0, 1, 2, 3
1	4, 5, 6, 7
2	8, 9, 10, 11

Escalonamento `dynamic`

UFABC

- As iterações também são quebradas em nacos (*chunks*) de tamanho `chunksize` consecutivas
- Cada thread executa um naco e quando uma thread termina um naco, ele requisita um novo para inicar o processamento ao ambiente de execução
- O processo se repete enquanto houver nacos disponíveis
- O valor `chunksize` pode ser omitido. Neste caso um valor padrão de 1 é utilizado.

Escalonamento `guided`

UFABC

- Assim como o dinâmico, cada thread também executa um naco e quando acaba requisita um novo ao sistema de execução.
- Contudo, no escalonador `guided` conforme os nacos são completados, o tamanho dos novos nacos diminuem.
- Se nenhum `chunksize` for especificado, o tamanho dos nacos diminui até 1.
- Se `chunksize` for especificado, o tamanho do naco diminui até `chunksize` com a exceção do último naco (que pode ser menor do que `chunksize`).

Exemplo: Aproximação Trapezoidal

UFABC

- Atribuição de iterações 1 a 9999 usando o escalonador `guided` com duas threads

Table 5.3 Assignment of Trapezoidal Rule Iterations 1–9999 using a `guided` Schedule with Two Threads

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1–5000	5000	4999
1	5001–7500	2500	2499
1	7501–8750	1250	1249
1	8751–9375	625	624
0	9376–9687	312	312
1	9688–9843	156	156
0	9844–9921	78	78
1	9922–9960	39	39
1	9961–9980	20	19
1	9981–9990	10	9
1	9991–9995	5	4
0	9996–9997	2	2
1	9998–9998	1	1
0	9999–9999	1	0

Escalonamento `runtime`

UFABC

- O sistema usa a variável de ambiente `OMP_SCHEDULE` para determinar, em tempo de execução, como fazer o escalonamento.
- A variável `OMP_SCHEDULE` pode receber qualquer um dos valores que puderem ser usados para `static`, `dynamic` ou `guided`