Instituto Politécnico do Cavado e Ave

Mestrado em Engenharia Informática

Programação Concorrente e Distribuída

2018/2019

Fernando Correia a11199

Conteúdo

Glossário

1. **Speedup** – é uma medida do grau de desempenho. O speedup mede o rácio entre o tempo de execução sequencial e o tempo de execução em paralelo.

2. **Lei de Amdahl** – dá-nos uma medida do speedup máximo que podemos obter ao resolver um determinado problema com p processadores, também pode ser utilizada para determinar o limite máximo de speedup que uma determinada aplicação poderá alcançar independentemente do número de processadores a utilizar.
3. **Thread** - é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrencialmente. O suporte à thread é fornecido pelo próprio sistema operacional no caso da linha de execução ao nível do núcleo KLT, ou implementada através de uma biblioteca de uma determinada linguagem, no caso de uma ULT. Uma thread permite, por exemplo, que o usuário de um programa utilize uma funcionalidade do ambiente enquanto outras linhas de execução realizam outros cálculos e operações.

4. **Programação concorrente** - um paradigma de programação para a construção de programas de computador que fazem uso da execução concorrente (simultânea) de várias tarefas computacionais interativas, que podem ser implementadas como programas separados ou como um conjunto de threads criadas por um único programa. Essas tarefas podem ser executadas por um único processador, vários processadores em um único equipamento ou processadores distribuídos por uma rede.

Introdução

Para a UC de Programação concorrente e distribuída do Mestrado em Engenharia Informática do Instituto Politécnico do Cavado e Ave, foi pedido pelo docente Nuno Lopes um trabalho para estimar o valor de PI pelos métodos Monte-Carlo e Gregory Leibniz através de uma grande quantidade de pontos / iterações de modo a poder obter resultados através de computação sequencial e paralela.

Segue em baixo todos os resultados.

Resolução do problema

1. Desenvolva uma aplicação sequencial (i.e., sem concorrência) para o cálculo de PI usando o método de Monte-Carlo que recebe como entrada a quantidade de pontos a gerar e produz como resultado o valor estimado de PI.

```
public static double formMontecarloSeq(long numeroPontos) {
    for (long i = 1; i <= numeroPontos; i++) {
        x = Math.random();
        y = Math.random();
        if (x * x + y * y <= 1)
            nSuccess++;
    }
    return (4.0* nSuccess / numeroPontos);
}
```

Código para a estimativa de PI utilizando o método de Monte-Carlo sequencial.

Numero de pontos parametrizável no main method:

```
public static void main(String[] args) {
    Scanner scannerObj = new Scanner(System.in);  // Create a Scanner
object
    System.out.print("Informe a quantidade de iterações a calcular:
");
    int numeroPontos = scannerObj.nextInt();
    System.out.println("N° Pontos : " + numeroPontos);  // Output user
input


    System.out.println("Numero de processadores: " +
Runtime.getRuntime().availableProcessors());
    //Thread[] listathreads = new Thread[2];
    long startTime = System.nanoTime();
    double result = formMontecarloSeq(numeroPontos);
    System.out.println("π: " + result);
    long stopTime = System.nanoTime();
    long totalTime = stopTime - startTime;
    System.out.println("Calculo sequencial demorou (secs): "  +
String.format("%.6f", (totalTime)/1.0e9) );

    long start = System.currentTimeMillis();
    for (int i = 0; i < ITERATIONS; ++i) {
        formMontecarloSeq(numeroPontos);
    }
    long elapsed = System.currentTimeMillis() - start;
    long average = elapsed / ITERATIONS;
    System.out.println("Para " + ITERATIONS + "iteracoes" + "o tempo
médio foi: " + String.format("%.6f", (average)/1.0e9) + "s");
}
```

2. Desenvolva uma aplicação sequencial (i.e., sem concorrência) para o cálculo de PI usando o método de Séries de Gregory-Leibniz que recebe como entrada a quantidade de iterações k a calcular e produz como resultado o valor estimado de PI.

```
public static double formCarlzSeq(int numeroIteracoes) {
    double factor = 1.0;
    double sum = 0.0;
    for (int k = 0; k < numeroIteracoes; k++) {
        sum += factor / (2 * k + 1);
        factor = -factor;
    }
    return (4.0 * sum);
}
```

Código para a estimativa de PI utilizando o método de método de Séries de Gregory-Leibniz sequencial.

Numero de iterações parametrizável no main method:

```
public static void main(String[] args) {
    Scanner scannerObj = new Scanner(System.in);
    System.out.print("Informe a quantidade de iterações a calcular:
");
    int numeroIteracoes = scannerObj.nextInt();
    System.out.println("N° Iterações : " + numeroIteracoes);


    System.out.println("Gregory Numero de processadores: " +
Runtime.getRuntime().availableProcessors());
    long startTime = System.nanoTime();
    double result = formCarlzSeq(numeroIteracoes);
    long stopTime = System.nanoTime();
    long totalTime = stopTime - startTime;
    System.out.println("π: " + result);
    System.out.println("Calculo Sequencial (secs): "  +
String.format("%.6f", (totalTime)/1.0e9) );

    long start = System.currentTimeMillis();
    for (int i = 0; i < ITERATIONS; ++i) {
        formCarlzSeq(numeroIteracoes);
    }
    long elapsed = System.currentTimeMillis() - start;
    long average = elapsed / ITERATIONS;
    System.out.println("Para " + ITERATIONS + " iteracoes" + "o tempo
médio foi: " + String.format("%.6f", (average)/1.0e9) + "s");
}
```

3. Apresente uma versão concorrente da aplicação com o método de Monte-Carlo, sendo que o programa deverá receber como entrada e produzir como resultado a mesma informação do programa sequencial. Adicionalmente, esta versão concorrente deve permitir também especificar como parâmetro de entrada o número de threads.

Para trabalhar com concorrência com este método usei os conceitos de "Atomic Integer", "Workers" e "ExecutorService".

Atomic Integers não são nada mais que um contador que se auto incrementa automaticamente.

```
@Override
    public void run() {
        double x = Math.random();
        double y = Math.random();
        if (x * x + y * y <= 1)
            nAtomSuccess.incrementAndGet();
    }
}
public MontecarloPar(int numeroPontos) {
    this.nAtomSuccess = new AtomicInteger(0);
    this.nThrows = numeroPontos;
    this.value = 0;
}
```

Se seguida usei a class "ExecuterService" para fazer a gestão de tarefas assíncronas. Uma vantagem é que podem ser terminadas a meio do processo.

```
public double getPi(int nProcessors) throws InterruptedException {
    //int nProcessors = Runtime.getRuntime().availableProcessors();
    ExecutorService executor =
Executors.newWorkStealingPool(nProcessors);
    Thread.sleep(3000);
```

Workers é um objecto que executa tarefas no background.

```
Runnable worker = new MonteCarlo();
executor.execute(worker);
```

Código para receber como parâmetro o numero de threads a utilizar:

```
System.out.println("N° threads : " + numetothreads);
//ThreadPoolExecutor executor = (ThreadPoolExecutor)
Executors.newFixedThreadPool(numetothreads);
MontecarloPar PiVal = new MontecarloPar(numeroPontos);

long startTime = System.currentTimeMillis();
double value = PiVal.getPi(numetothreads);
```

4. MONTEParalel_BETA

Segue neste trabalho uma tentativa de fazer uma outra abordagem concorrencial para o Método de Monte Carlo através de Executers e Futures de modo a poder partir o problema em várias partes e depois juntar o resultado. Está incompleto mas o objectivo passaria por conseguir meter mais inputs de forma a obter um resultado mais próximo.

5. Apresente uma versão concorrente da aplicação com o método de Séries de Gregory-Leibniz, sendo que o programa deverá receber como entrada e produzir como resultado a mesma informação do programa sequencial. Adicionalmente, esta versão concorrente deve permitir também especificar como parâmetro de entrada o número de threads.

Para este problema decidir fazer a mesma abordagem do problema anterior mas partindo os blocos em quatro partes, ou seja, numa amostra de 2000000000 fiz:

```
//separa o cálculo em 4 partes definindo o valor de n inicial e final
para cada uma
Future<Double> parte1 = es.submit(new CarlzParalel(1, 300000000));
Future<Double> parte2 = es.submit(new CarlzParalel(300000001,
600000000));
Future<Double> parte3 = es.submit(new CarlzParalel(600000001,
900000000));
Future<Double> parte4 = es.submit(new CarlzParalel(900000001,
1200000000));
```

Qualquer um destes algoritmos está com os parâmetros que o professor pede no enunciado seja input de threads, amostras ou iterações tal como pode ver no código e nas explicações mais à frente deste relatório.

Analise do Desempenho

1. De modo a medir o desempenho das implementações anteriores, acrescente no código das 4 aplicações anteriores um mecanismo de medição do tempo de execução

    - durante a medição do tempo de execução não deve haver qualquer output para a consola

Todo o input é feito antes de ser feito qualquer calculo. Segue abaixo o exemplo do Monte-Carlo Paralelo:

```
Scanner scannerObj = new Scanner(System.in);  // Create a Scanner
object
System.out.print("Informe a quantidade de iterações a calcular: ");
int numeroPontos = scannerObj.nextInt();
System.out.println("N° Pontos : " + numeroPontos);  // Output user
input
System.out.print("Informe a número máx de threads:");
int numetothreads = scannerObj.nextInt();
System.out.println("N° threads : " + numetothreads);
//ThreadPoolExecutor executor = (ThreadPoolExecutor)
Executors.newFixedThreadPool(numetothreads);
MontecarloPar PiVal = new MontecarloPar(numeroPontos);

long startTime = System.currentTimeMillis();
double value = PiVal.getPi(numetothreads);
long stopTime = System.currentTimeMillis();
```

    - o cálculo do tempo de execução na versão paralela deverá incluir a criação das threads.

```
System.out.println("N° threads : " + numetothreads);
//https://stackoverflow.com/questions/949355/executors-
newcachedthreadpool-versus-executors-newfixedthreadpool
ExecutorService es = Executors.newFixedThreadPool(numetothreads);
//separa o cálculo em 4 partes definindo o valor de n inicial e final
para cada uma
```

    - a recolha do tempo de execução de cada aplicação deve ser uma média de várias execuções, configurável (por exemplo: 10), em vez de uma única execução.

```java
private final static int ITERATIONS = 10;
```

```java
for (int i = 0; i <= ITERATIONS; ++i) {
    parte1 = es.submit(new CarlzParalel(1, 500000000));
    parte2 = es.submit(new CarlzParalel(500000001, 1000000000));
    parte3 = es.submit(new CarlzParalel(1000000001, 1500000000));
    parte4 = es.submit(new CarlzParalel(1500000001, 2000000000));
}
```

```java
private final static int ITERATIONS = 10;
```

```java
for (int i = 0; i < ITERATIONS; ++i) {
    formMontecarloSeq(numeroPontos);
}
long elapsed = System.currentTimeMillis() - start;
long average = elapsed / ITERATIONS;
System.out.println("Para " + ITERATIONS + "iteracoes" + "o tempo médio
foi: " + String.format("%.6f", (average)/1.0e9) + "s");
```

Como se pode ver por estes dois exemplos, o numero de ITERATIONS é um parâmetro parametrizável no inicio do programa.

Optei por introduzir uma variável no programa para evitar mais inputs do utilizador. Está assim para os 4 problemas!

2. Com base nos programas desenvolvidos nos pontos anteriores, construa uma tabela e um gráfico onde representa no eixo dos YY o tempo de execução do programa, e no eixo dos XX o nº de threads (assuma como exemplo: sequencial, PC1, PC2, PC4, PC8, PC10, PC20)

O valor escolhido para demorar pelo menos um minuto foi 2000000000. As durações são dadas em segundos.

Todos os resultados podem ser consultados no ficheiro Excel que vai dento do zip.

Tal como já foi descrito em cima, Para Monte-Carlo Paralelo não usei os mesmo valores devido a problemas técnicos. Usarei o valor 40000000 para o numero de iterações do método Monte-Carlo com Paralelismo e para manter coerência nos dados tenho duas tabelas, uma com o sequencial com 40000000 para poder fazer comparações e outra com 2000000000.

Ao tentar implementar o método Monte-Carlo paralelo com 2000000000 obtinha a seguinte mensagem de erro:

Exception in thread "main" java.util.concurrent.RejectedExecutionException: Queue capacity exceeded

at java.util.concurrent.ForkJoinPool$WorkQueue.growArray(ForkJoinPool.java:884)

at java.util.concurrent.ForkJoinPool.externalSubmit(ForkJoinPool.java:2357)

at java.util.concurrent.ForkJoinPool.externalPush(ForkJoinPool.java:2419)

at java.util.concurrent.ForkJoinPool.execute(ForkJoinPool.java:2648)

at MontecarloPar.getPi(MontecarloPar.java:36)

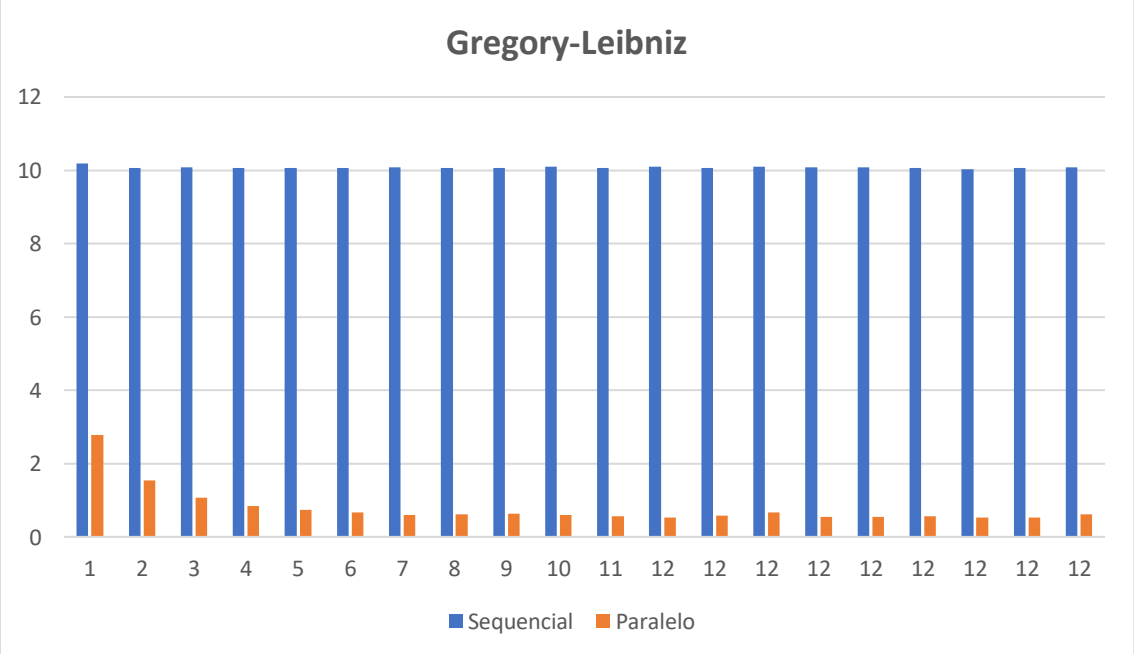
at MontecarloPar.main(MontecarloPar.java:68)

```
Tentei implementar o método Monte-Carlo Paralelo com a mesta técnica
de divisão por partes mas infelizmente para o numero 2000000000 o
intervalo de tempo de execução estava acima dos 1000s e não tive tempo
para optimizar o algoritmo pelo que deixei noutro ficheiro como nome
BETA. MONTEParalel_BETA
```

Calculo concorrencial (secs): 1096,989442

Para o método de Gregory-Leibniz temos os seguintes valores:

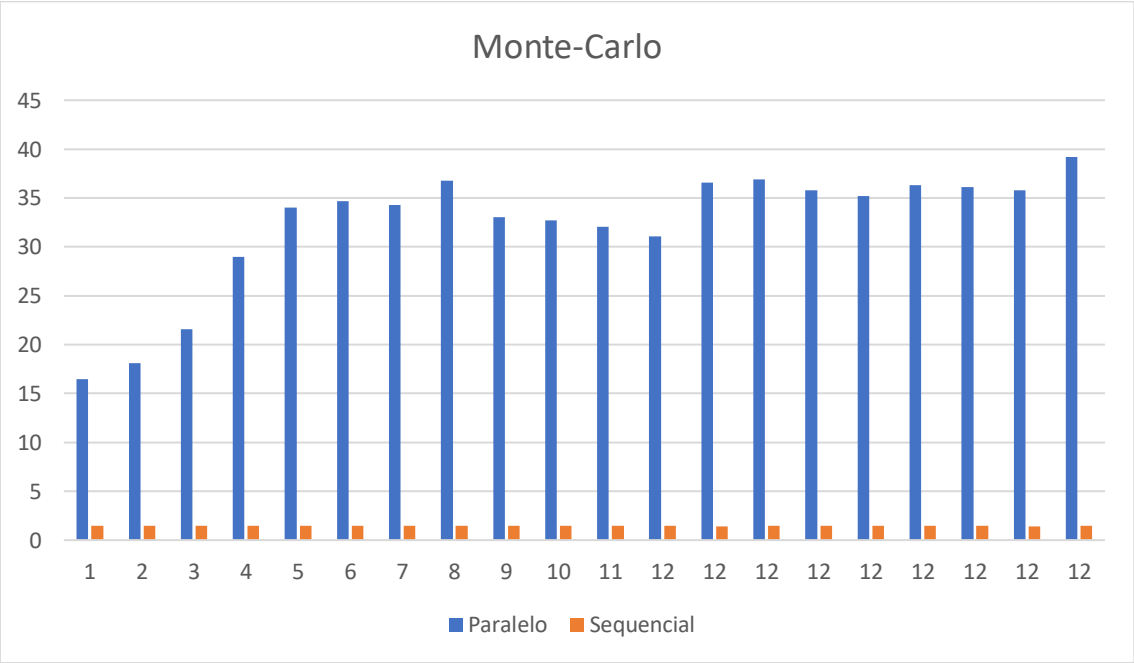
Numero de threads	Sequencial	Numero de threads	Paralelismo	Diferença dos tempos de execu	Speedup	% Sequential code	Numero de Iterações	Paralel Efficiency	Sequencial Effi
12	10,1880028	1	2,77837509	-7,40962771	3,666892507	#DIV/0!	2000000000	3,666892507	0,305574376
12	10,06363822	2	1,54810443	-8,51553379	6,500619742	-0,692337026	2000000000	3,250309871	0,541718312
12	10,08337117	3	1,07242637	-9,0109448	9,40239018	-0,340466097	2000000000	3,13413006	0,783532515
12	10,06730806	4	0,84226054	-9,22504752	11,95272434	-0,221782753	2000000000	2,988181086	0,996060362
12	10,06799295	5	0,73071924	-9,33727371	13,77819606	-0,159276948	2000000000	2,755639211	1,148183005
12	10,05950456	6	0,66507048	-9,39443408	15,12547145	-0,12066363	2000000000	2,520911909	1,260455954
12	10,08219828	7	0,60510188	-9,4770964	16,66198472	-0,096646997	2000000000	2,380283532	1,388498727
12	10,06673528	8	0,60850521	-9,45823007	16,54338388	-0,073774714	2000000000	2,067922985	1,378615323
12	10,06853687	9	0,63050122	-9,43803565	15,96909974	-0,054551445	2000000000	1,774344416	1,330758312
12	10,09069728	10	0,6033543	-9,48734298	16,72433143	-0,044674308	2000000000	1,672433143	1,393694285
12	10,06701743	11	0,55411253	-9,5129049	18,16782131	-0,03945339	2000000000	1,651620119	1,513985109
12	**10,10962878**	**12**	**0,59556074**	**-9,51406804**	16,97497518	-0,026643366	**2000000000**	**1,414581265**	**1,414581265**
12	10,06357608	12	0,5708283	-9,49274778	17,62977778	-0,029030314	2000000000	1,469148148	1,469148148
12	10,09021973	12	0,66242647	-9,42779326	15,23221095	-0,019290526	2000000000	1,269350913	1,269350913
12	10,0846619	12	0,54723835	-9,53742355	18,42828066	-0,03171154	2000000000	1,535690055	1,535690055
12	10,07932651	12	0,54094481	-9,5383817	18,63281859	-0,032361368	2000000000	1,552734882	1,552734882
12	10,07107423	12	0,56954464	-9,50152959	17,68267757	-0,029215431	2000000000	1,473556464	1,473556464
12	10,03657182	12	0,53505354	-9,50151828	18,75807012	-0,032752304	2000000000	1,56317251	1,56317251
12	**10,06763238**	**12**	**0,52998855**	**-9,53764383**	18,99594318	-0,033480561	**2000000000**	**1,582995265**	**1,582995265**
12	10,07482621	12	0,61521534	-9,45961087	16,37609721	-0,024293152	2000000000	1,364674767	1,364674767



Ter em atenção que Sequencial usou o número total de threads e paralelo usou sempre numero diferente de threads (1 a 12, e depois sempre 12). As conclusões tiradas é que depois de usar um número superior a 4 threads, os valores mantem-se constantes mas sempre decrescentes.

Para o método de Monte-Carlo temos os seguintes valores:

Numero de threads	Sequencial	Numero de threads	Paralelismo	Diferença dos tempos de execu	Speedup	% código se	Numero de Iterações	Paralel Efficie	Sequencia
12	1,36760612	1	15,13375179	13,76614567	0,09036795	#DIV/0!	40000000	0,09036795	0,00753066
12	1,40774175	2	25,97876432	24,57102257	0,054188172	35,90842347	40000000	0,027094086	0,00451568
12	1,39760612	3	28,31861666	26,92101054	0,049352909	29,89334501	40000000	0,01645097	0,00411274
12	1,39992779	4	31,73418123	30,33425344	0,044114193	29,89125535	40000000	0,011028548	0,00367618
12	1,46359693	5	36,36691899	34,90332206	0,040245283	30,80954092	40000000	0,008049057	0,00335377
12	1,40639061	6	34,40183705	32,99544644	0,040881265	29,15329927	40000000	0,006813544	0,00340677
12	1,39717023	7	36,36568531	34,96851508	0,038420017	30,19944904	40000000	0,005488574	0,00320167
12	1,43959537	8	36,77851533	35,33891996	0,039142292	29,05464503	40000000	0,004892786	0,00326186
12	1,442754259	9	33,05462063	31,61186637	0,043647582	25,64962376	40000000	0,004849731	0,0036373
12	1,441731669	10	32,72611718	31,28438551	0,044054468	25,11019269	40000000	0,004405447	0,00367121
12	1,44351189	11	32,07181722	30,62830533	0,045008734	24,33970097	40000000	0,004091703	0,00375073
12	1,44425294	12	31,0983761	29,65412316	0,046441426	23,39909066	40000000	0,003870119	0,00387012
12	1,42999917	12	36,60125416	35,17125499	0,039069677	27,83123362	40000000	0,003255806	0,00325581
12	1,4473313	12	36,88753154	35,44020024	0,039236328	27,71263768	40000000	0,003269694	0,00326969
12	1,43432539	12	35,81263816	34,37831277	0,040050816	27,14721471	40000000	0,003337568	0,00333757
12	1,4372625	12	35,1827391	33,7454766	0,040851353	26,61344723	40000000	0,003404279	0,00340428
12	1,43222206	12	36,28101921	34,84879715	0,039475795	27,54397714	40000000	0,00328965	0,00328965
12	1,4340416	12	36,09261824	34,65857664	0,039732269	27,36559242	40000000	0,003311022	0,00331102
12	1,4025635	12	38,65932445	37,25676095	0,036280083	29,97818118	40000000	0,00302334	0,00302334
12	1,43764017	12	39,1771066	37,73946643	0,036695925	29,63743506	40000000	0,003057994	0,00305799



Os valores sequenciais para este resultado foram muito pequenos pelo que resolvi só implementar para o sequencial uma amostra maior de 2000000000.

Escolhi os valores máximos que me permitiram calcular ambos os casos sequencial e paralelo, infelizmente não consegui chegar a um minuto de tempo pois eu ficava sem memória antes ou a JVM gerava erro.

3. Compare o desempenho da execução concorrente com a execução sequencial, calculando o valor de Alfa (percentagem de código sequencial do programa) de acordo com a Lei de Amdahl, tomando como referência os ganhos de desempenho obtidos. Indique o tempo de execução medido para os casos sequencial e concorrente. Faça o cálculo para a melhor das versões concorrente (indicando o nº de threads assumido).

Segue abaixo os resultados para ambos os métodos.

Speedup = (Valor Paralelo / Valor Sequencial)

Percentagem sequencial = ((nº cores / Speedup)- 1 / (nº cores-1))

Para os cálculos da percentagem sequencial foram considerados o numero de threads usados no paralelismo, ou seja, o numero de threads usados nas sequencias paralelas.

Eficiência Paralela = Speedup / nº cores execução paralela

Eficiência Sequencial = Speedup / nº cores execução sequencial

Tudo calculado no código

É possível verificar que no método Monte-Carlo seria possível meter mais pontos mas como não consegui resolver a questão da concorrência da melhor maneira, a JVM gerava um erro pelo que tive que introduzir os pontos máximos para conseguir executar o programa.

Método Gregory-Leibniz

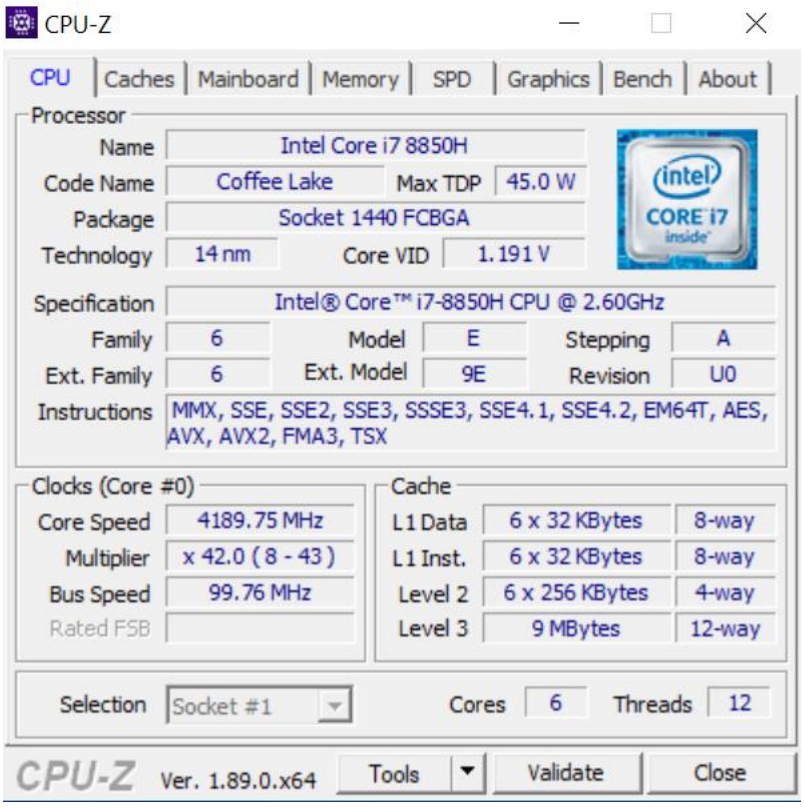
Numero de threads	Sequencial	Numero de threads	Paralelismo	Diferença dos tempos de execu	Speedup	% Sequential code	Numero de Iterações	Paralel Efficiency	Sequencial Effi
12	10,1880028	1	2,77837509	-7,40962771	3,666892507	#DIV/0!	2000000000	3,666892507	0,305574376
12	10,06363822	2	1,54810443	-8,51553379	6,500619742	-0,692337026	2000000000	3,250309871	0,541718312
12	10,08337117	3	1,07242637	-9,0109448	9,40239018	-0,340466097	2000000000	3,13413006	0,783532515
12	10,06730806	4	0,84226054	-9,22504752	11,95272434	-0,221782753	2000000000	2,988181086	0,996060362
12	10,06799295	5	0,73071924	-9,33727371	13,77819606	-0,159276948	2000000000	2,755639211	1,148183005
12	10,05950456	6	0,66507048	-9,39443408	15,12547145	-0,12066363	2000000000	2,520911909	1,260455954
12	10,08219828	7	0,60510188	-9,4770964	16,66198472	-0,096646997	2000000000	2,380283532	1,388498727
12	10,06673528	8	0,60850521	-9,45823007	16,54338388	-0,073774714	2000000000	2,067922985	1,378615323
12	10,06853687	9	0,63050122	-9,43803565	15,96909974	-0,054551445	2000000000	1,774344416	1,330758312
12	10,09069728	10	0,6033543	-9,48734298	16,72433143	-0,044674308	2000000000	1,672433143	1,393694285
12	10,06701743	11	0,55411253	-9,5129049	18,16782131	-0,03945339	2000000000	1,651620119	1,513985109
12	**10,10962878**	**12**	**0,59556074**	**-9,51406804**	16,97497518	-0,026643366	**2000000000**	**1,414581265**	**1,414581265**
12	10,06357608	12	0,5708283	-9,49274778	17,62977778	-0,029030314	2000000000	1,469148148	1,469148148
12	10,09021973	12	0,66242647	-9,42779326	15,23221095	-0,019290526	2000000000	1,269350913	1,269350913
12	10,0846619	12	0,54723835	-9,53742355	18,42828066	-0,03171154	2000000000	1,535690055	1,535690055
12	10,07932651	12	0,54094481	-9,5383817	18,63281859	-0,032361368	2000000000	1,552734882	1,552734882
12	10,07107423	12	0,56954464	-9,50152959	17,68267757	-0,029215431	2000000000	1,473556464	1,473556464
12	10,03657182	12	0,53505354	-9,50151828	18,75807012	-0,032752304	2000000000	1,56317251	1,56317251
12	**10,06763238**	**12**	**0,52998855**	**-9,53764383**	18,99594318	-0,033480561	**2000000000**	**1,582995265**	**1,582995265**
12	10,07482621	12	0,61521534	-9,45961087	16,37609721	-0,024293152	2000000000	1,364674767	1,364674767

Método Monte-Carlo

Numero de threads	Sequencial	Numero de threads	Paralelismo	Diferença dos tempos de execu	Speedup	% código se	Numero de Iterações	Paralel Efficie	Sequencia
12	1,36760612	1	15,13375179	13,76614567	0,09036795	#DIV/0!	40000000	0,09036795	0,00753066
12	1,40774175	2	25,97876432	24,57102257	0,054188172	35,90842347	40000000	0,027094086	0,00451568
12	1,39760612	3	28,31861666	26,92101054	0,049352909	29,89334501	40000000	0,01645097	0,00411274
12	1,39992779	4	31,73418123	30,33425344	0,044114193	29,89125535	40000000	0,011028548	0,00367618
12	1,46359693	5	36,36691899	34,90332206	0,040245283	30,80954092	40000000	0,008049057	0,00335377
12	1,40639061	6	34,40183705	32,99544644	0,040881265	29,15329927	40000000	0,006813544	0,00340677
12	1,39717023	7	36,36568531	34,96851508	0,038420017	30,19944904	40000000	0,005488574	0,00320167
12	1,43959537	8	36,77851533	35,33891996	0,039142292	29,05464503	40000000	0,004892786	0,00326186
12	1,442754259	9	33,05462063	31,61186637	0,043647582	25,64962376	40000000	0,004849731	0,0036373
12	1,441731669	10	32,72611718	31,28438551	0,044054468	25,11019269	40000000	0,004405447	0,00367121
12	1,44351189	11	32,07181722	30,62830533	0,045008734	24,33970097	40000000	0,004091703	0,00375073
12	1,44425294	12	31,0983761	29,65412316	0,046441426	23,39909066	40000000	0,003870119	0,00387012
12	1,42999917	12	36,60125416	35,17125499	0,039069677	27,83123362	40000000	0,003255806	0,00325581
12	1,4473313	12	36,88753154	35,44020024	0,039236328	27,71263768	40000000	0,003269694	0,00326969
12	1,43432539	12	35,81263816	34,37831277	0,040050816	27,14721471	40000000	0,003337568	0,00333757
12	1,4372625	12	35,1827391	33,7454766	0,040851353	26,61344723	40000000	0,003404279	0,00340428
12	1,43222206	12	36,28101921	34,84879715	0,039475795	27,54397714	40000000	0,00328965	0,00328965
12	1,4340416	12	36,09261824	34,65857664	0,039732269	27,36559242	40000000	0,003311022	0,00331102
12	1,4025635	12	38,65932445	37,25676095	0,036280083	29,97818118	40000000	0,00302334	0,00302334
12	1,43764017	12	39,1771066	37,73946643	0,036695925	29,63743506	40000000	0,003057994	0,00305799

- Indique o nº de processadores presentes na máquina usada para os resultados.

Para estes resultados foi utilizada a seguinte configuração:



4. Comente os resultados dos tempos de execução para as versões sequencial e concorrentes, nos dois métodos da aplicação, apresentando justificações para os valores obtidos.

Das vinte execuções das quais os resultados estão na tabela anterior assim como retratadas no gráfico acima.
Podemos concluir que é de facto uma vantagem tirar partido da programação em paralelo isto é usar threads, e colocar cada uma a fazer tarefas independentes sendo que desta forma a rapidez de execução de um programa será de fato aumentada conforme podemos constatar pelos resultados obtidos.
Podemos também ressalvar que criar um programa em paralelo e usar uma única threads, não faz sentido pois será mais lento do que o mesmo programa em sequencial e desta forma não sendo possível calcular o Alfa pois é impossível aplicar a Lei de Amdahl, visto que teríamos de dividir por zero sendo que isso não é matemática possível.
O desenvolvido programa é determinístico, isto é, obtém-se sempre o resultado esperado.
Constata-se que aumentado o número de threads a rapidez é maior obtendo-se assim um speedUp, ou seja, um ganho maior, conforme se pode constatar nos resultados obtidos. Não adianta juntar cores para obter maior velocidade de processos mas pode-se hoje em dia adoptar técnicas para melhorar os processos em execução.

Tive pena por só conseguir usar 40000000 no método Monte-Carlo pois acho que com valores superiores iria obter resultados mais interessantes!