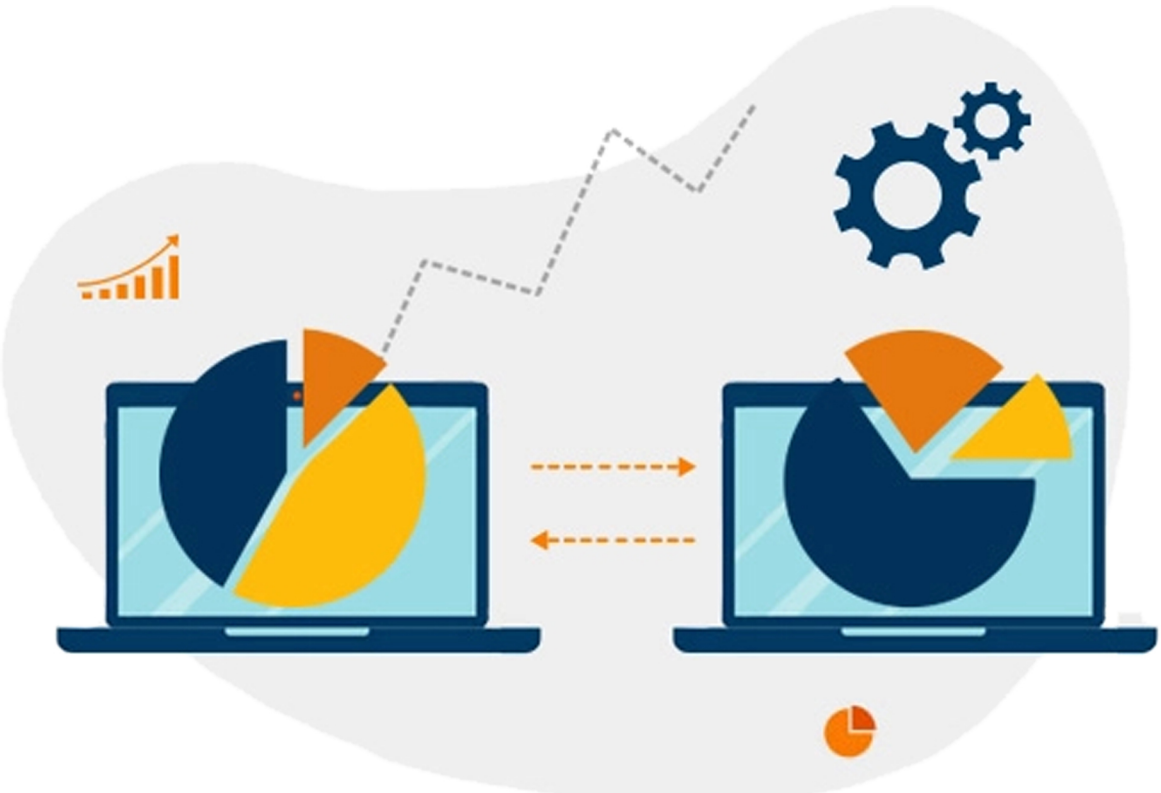
REPORT SPPEDUP

UC: DISTRIBUIÇAO E INTEGRAÇAO DE SITEMAS
DOCENTE: PROF. DRA. FERNANDA PASSOS



Trabalho realizado por:
Ivo Nunes nº20202567 2ºano GSC

ATLÂNTICA
INSTITUTO UNIVERSITÁRIO

Índice

Contexto **3**

Análise Sequencial **5**

Análise Concorrente **6**

Análise Paralelo **9**

Conclusão **11**

Contexto

A redução do tempo de execução do programa é uma das vantagens que os programadores de aplicações esperam alcançar para aumentar a sua eficiência. A conversão de programas sequenciais em programas paralelos ou concorrentes é um dever dispendioso, já que requerem equipamentos de hardware ou software poderosos. É preferível antecipar virtualmente a velocidade ganha do paralelismo ou da concorrência, antes de executar a aplicação num ambiente paralelo real.

Vários sistemas foram desenvolvidos para analisar o desempenho de programas paralelos e concorrentes, no entanto neste relatório tenciono efetuar a minha análise pessoal, face ao problema colocado na UC de Distribuição e Integração de Sistemas, lecionada pela Professora Doutora Fernanda Passos.

Explicando sucintamente, os programas consistem em introduzir um número N, maior que 0, e o programa adicionar todos os números primos dentro de uma lista e obter os tempos de computação deste processo. Teremos seis programas cada um com o seu paradigma, ou seja, sequencial, concorrente e paralelo. Posteriormente, iremos realizar uma análise concreta de todos os resultados.

Se a mesma largura de banda de memória estiver disponível num ou mais processos, verificamos praticamente nenhum aumento de velocidade, uma vez que a SpMV (*Sparse matrix–vector multiplication*) e as operações de álgebra linear relacionadas são limitadas pela largura de banda de memória.

Pode também acontecer que a sobrecarga de comunicação ultrapasse o seu cálculo local. Por exemplo, nos métodos iterativos lineares recomenda-se pelo menos 10.000 incógnitas por processo ou fatores numéricos.

Posto isto, realizou-se seis programas distintos, sendo três menos eficientes e três bastante mais eficientes. Um número é primo quando é divisível apenas por ele próprio e 1. Realizou-se um programa que percorre os números todos até ao N digitado, adicionando os números primos numa lista. No final, se o número indicado estiver dentro dessa lista, ele é primo. No mais eficiente fomos pela divisibilidade, já que no programa, se o número for primo, só utiliza a memória para armazenar dois valores, o 1 e ele próprio. Depois, verificando a len da lista, retornamos se N é primo ou não, pois sabemos que para ser primo a len tem de, obrigatoriamente, ser igual a dois.

Esta experiência foi efetuada num ambiente com o sistema operativo macOS Monterey, com um processador Apple M1 (arquitetura ARM), contendo 8 núcleos (4 performance e 4 de eficiência), 8GB de RAM e executado na plataforma VS Code e com a versão de Python 3.10.

Análise Sequencial

Como abordado anteriormente, em cada análise iremos ter dois resultados de dois programas, um mais eficiente e um menos eficiente. Na realização dos códigos de processamento sequencial, que se encontram em dois ficheiros em anexo, obteve-se a seguinte tabela de resultados:

PROGRAMAS MENOS EFICIENTES			
SEQUENCIAL			
	Tentativa	Tempo Sequencial (Segundos)	N
0	1	11,68255305	93169
1	2	11,67096281	93169
2	3	11,62094903	93169
3	4	11,64784527	93169
4	5	11,64039111	93169
5	6	11,72094917	93169
6	7	11,77627015	93169
7	8	11,62498999	93169
8	9	11,62539697	93169
9	10	11,63260412	93169
10	11	11,61280084	93169
11	12	11,73855805	93169
12	13	11,655617	93169
13	14	11,58415627	93169
14	15	11,66603971	93169
AVG:		11,66000557	

PROGRAMAS MAIS EFICIENTES			
SEQUENCIAL			
	Tentativa	Tempo Sequencial (Segundos)	N
15	16	11,2295	768261929
16	17	11,4278	768261929
17	18	11,3796	768261929
18	19	11,4011	768261929
19	20	11,3796	768261929
20	21	11,3826	768261929
21	22	11,4186	768261929
22	23	11,4418	768261929
23	24	11,425	768261929
24	25	11,3886	768261929
25	26	11,3807	768261929
26	27	11,4592	768261929
27	28	11,4052	768261929
28	29	11,3857	768261929
29	30	11,4289	768261929
AVG:		11,39559333	

Analisando os tempos, podemos achá-los idênticos, mas analisando com atenção, verificamos que o número N é extremamente superior no programa mais eficiente, utilizando N = 93169. O tempo de execução do programa sequencial seria de 0.005269289016723633 segundos. Estamos a falar de uma execução, aproximadamente, 221610 % mais rápida.

Finalizando a análise, ambos os programas atingiram a média de, aproximadamente, 11 segundos.

Análise Concorrente

Neste paradigma realizaram-se 3 testes distintos, com recurso a 2, 4 e 6 *Threads*, para podermos verificar se com o aumento das mesmas traria algum impacto ao programa na redução de tempo de execução do mesmo.

PROGRAMAS MENOS EFICIENTES

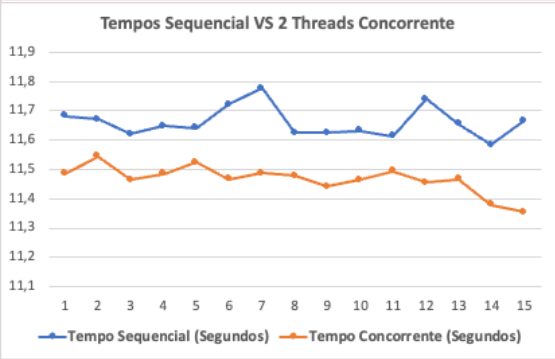
CONCORRENTE				
	Tentativa	Tempo Concorrente (Segundos)	Threads	N
0	1	11,48379493	2	93169
1	2	11,54376817	2	93169
2	3	11,46311975	2	93169
3	4	11,48391604	2	93169
4	5	11,52455878	2	93169
5	6	11,46716213	2	93169
6	7	11,48770809	2	93169
7	8	11,47690797	2	93169
8	9	11,44226313	2	93169
9	10	11,46397686	2	93169
10	11	11,49312997	2	93169
11	12	11,45426893	2	93169
12	13	11,46591902	2	93169
13	14	11,3781271	2	93169
14	15	11,35423684	2	93169
15	16	11,3531661	4	93169
16	17	11,35550117	4	93169
17	18	11,37493682	4	93169
18	19	11,37143493	4	93169
19	20	11,35566211	4	93169
20	21	11,35045028	4	93169
21	22	11,37221289	4	93169
22	23	11,37918496	4	93169
23	24	11,371979	4	93169
24	25	11,63931203	4	93169
25	26	11,47024179	4	93169
26	27	11,49592805	4	93169
27	28	11,69105315	4	93169
28	29	11,53982377	4	93169
29	30	11,51440382	4	93169
30	31	11,54330397	6	93169
31	32	11,51681113	6	93169
32	33	11,50540185	6	93169
33	34	11,46196389	6	93169
34	35	11,6390729	6	93169
35	36	11,51582503	6	93169
36	37	11,46656609	6	93169
37	38	11,46978688	6	93169
38	39	11,53874588	6	93169
39	40	11,52458334	6	93169
40	41	11,55240798	6	93169
41	42	11,51953292	6	93169
42	43	11,50993896	6	93169
43	44	11,50872102	6	93169
44	45	11,47479105	6	93169
AVG 2 Threads:		11,46552385		
AVG 4 Threads:		11,44235272		
AVG 6 Threads:		11,51649686		

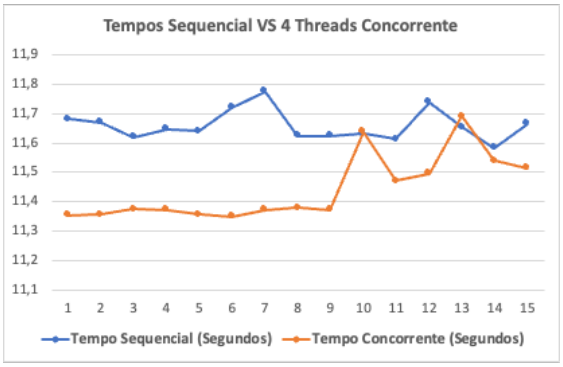
PROGRAMAS MAIS EFICIENTES

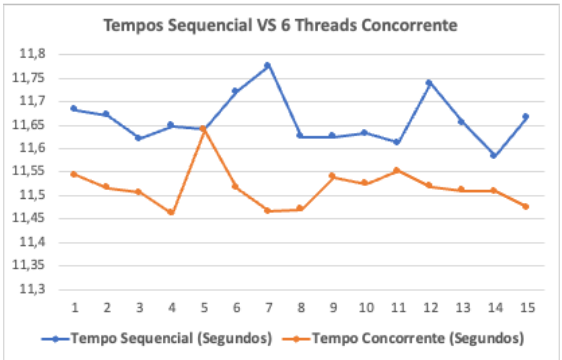
CONCORRENTE				
	Tentativa	Tempo Concorrente (Segundos)	Threads	N
45	46	12,67045784	2	768261929
46	47	12,63345408	2	768261929
47	48	12,59193492	2	768261929
48	49	12,62935424	2	768261929
49	50	12,62393379	2	768261929
50	51	12,68838	2	768261929
51	52	12,65793419	2	768261929
52	53	12,62768388	2	768261929
53	54	12,63030005	2	768261929
54	55	12,58595395	2	768261929
55	56	12,68475795	2	768261929
56	57	12,61426997	2	768261929
57	58	12,66830635	2	768261929
58	59	12,6071713	2	768261929
59	60	12,64328504	2	768261929
60	61	12,70884824	4	768261929
61	62	12,61767387	4	768261929
62	63	12,66983914	4	768261929
63	64	13,14466715	4	768261929
64	65	12,61730909	4	768261929
65	66	12,69019103	4	768261929
66	67	12,58158398	4	768261929
67	68	12,61136889	4	768261929
68	69	12,63801503	4	768261929
69	70	12,60759306	4	768261929
70	71	12,64924312	4	768261929
71	72	13,08971524	4	768261929
72	73	13,39191294	4	768261929
73	74	12,61534905	4	768261929
74	75	12,6862042	4	768261929
75	76	12,64085507	6	768261929
76	77	12,62638211	6	768261929
77	78	12,61944699	6	768261929
78	79	12,63253093	6	768261929
79	80	12,71110201	6	768261929
80	81	12,6359508	6	768261929
81	82	12,61525583	6	768261929
82	83	12,62330008	6	768261929
83	84	12,61896276	6	768261929
84	85	12,63113308	6	768261929
85	86	12,6286211	6	768261929
86	87	12,65893197	6	768261929
87	88	12,61897206	6	768261929
88	89	13,08592796	6	768261929
89	90	12,71336484	6	768261929
AVG 2 Threads:		12,63714517		
AVG 4 Threads:		12,75463427		
AVG 6 Threads:		12,67071584		

Como podemos observar, em cada uma das possibilidades realizou-se novamente um conjunto de 15 tentativas da qual foi calculada uma média precisamente igual ao exemplo sequencial.

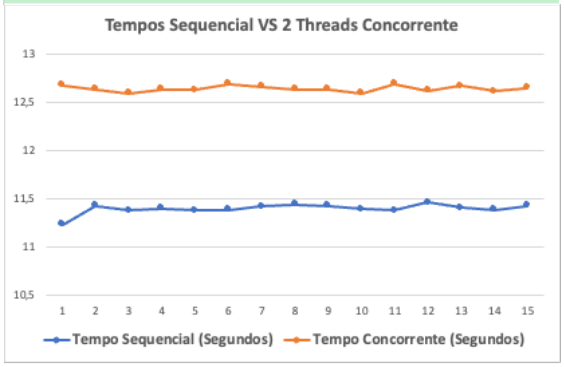
PROGRAMAS MENOS EFICIENTES

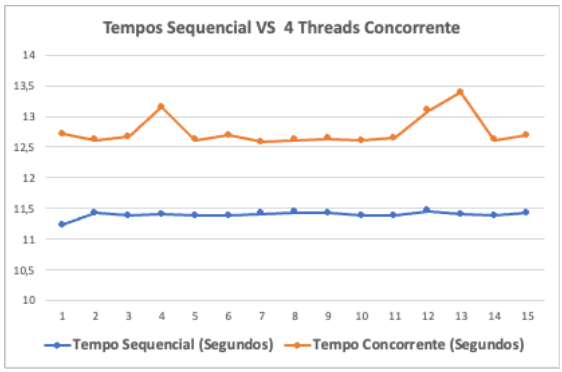


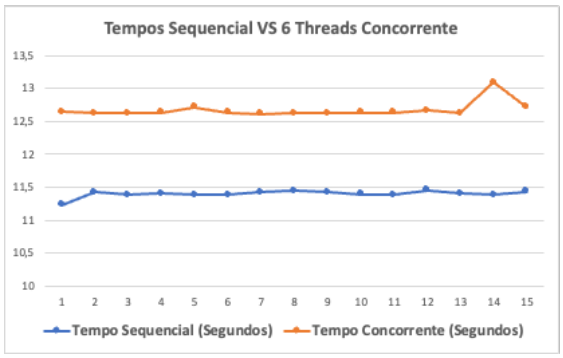




PROGRAMAS MAIS EFICIENTES







Ao analisarmos os resultados nos gráficos acima e sabendo que os primeiros programas mesmo sendo menos eficientes acedem mais vezes à memória, podemos concluir que, acedendo mais vezes à memória a velocidade de execução do programa tem tendência a ser menor que a do sequencial, tendo uma diferença apenas de décimas de segundo. No programa otimizado a diferença chega a ser de 2 segundos a mais que o seu sequencial.

De seguida, efetuou-se o cálculo do *SpeedUP*, que é nada mais nada menos que a velocidade média do programa sequencial a dividir pelo tempo demorado pelo concorrente.

Obtiveram-se assim os seguintes resultados:

PROGRAMAS MENOS EFICIENTES		
Speed UP CONCORRENTE		
	Resultado	Percentagem %
Resultado 2 Threads:	1,01696231	102%
Resultado 4 Threads:	1,01902169	102%
Resultado 6 Threads:	1,01246114	101%

PROGRAMAS MAIS EFICIENTES		
Speed UP CONCORRENTE		
	Resultado	Percentagem %
Resultado 2 Threads:	0,90175377	90%
Resultado 4 Threads:	0,89344728	89%
Resultado 6 Threads:	0,8993646	90%

Como já observado anteriormente, existindo mais acessos à memória, tivemos um ganho de tempo de 1,25 % a 1,90 % de tempo face ao sequencial. Apesar de pouco significativo, no programa mais eficiente, que não utilizada tanta memória, podemos verificar um aumento nos tempos, que chega a ser menos de 10,06%.

Em suma, de facto o *multi-thread* pode ser mais lento devido à sobrecarga de criação das *Threads*, da mudança de contexto entre elas e a competição pelos recursos da máquina. O programa *multi threaded* teve um desempenho pior devido à sobrecarga na criação das *threads* e forçando-as a esperar com o *mutex*. No entanto, no primeiro caso, devido a estar sempre a aceder à memória, tornou-o um pouco mais rápido pois a competição entre recursos é favorável neste quesito.

Análise Paralelo

Neste paradigma realizaram-se 3 testes distintos, com recurso a 2, 4 e 6 processos respetivamente, para verificarmos se com o aumento dos processos existiria algum impacto no programa, relativamente à redução de tempo de execução do mesmo.

Ao analisarmos os resultados nos gráficos acima, e sabendo que os primeiros programas mesmo sendo menos eficientes acedem mais vezes à memória, podemos concluir que acedendo mais vezes à memória a velocidade de execução do programa tem tendência a manter-se similar ao programa sequencial, mesmo sendo facilmente observada uma redução de, aproximadamente, 3 segundos. No caso mais eficiente, a diferença cai como esperado para quase metade, que era o objetivo principal do *SpeedUP,* com a criação de mais um processo, obtendo assim metade do tempo de processamento. Estas diferenças continuam a ser facilmente detetáveis com 4 e 6 processos, no entanto com 4 processos o programa menos eficiente atinge

finalmente o *SpeedUP* esperado com 2, enquanto o mais eficiente com 4 processos atinge a marca de mais de 1/3 do tempo sequencial, o que continua a ser um resultado bastante satisfatório, atingindo o resultado esperado, já que deveria ser, aproximadamente, 4 vezes mais rápido. Relativamente aos 6 processos, no menos eficiente, a marca chega aos 4,49 segundos e o mais eficiente aos 3,35 segundos, denotando-se que estamos a chegar ao "limite" de *cores* da máquina utilizada.

Depois efetuou-se o cálculo do *SpeedUP,* que é nada mais nada menos que a velocidade média do programa sequencial, a dividir pelo tempo demorado pelo paralelo.

Obtiveram-se assim os seguintes resultados:

PROGRAMAS MENOS EFICIENTES		
Speed UP PARALELO		
	Resultado	Percentagem %
Resultado 2 Processos:	1,28950677	129%
Resultado 4 Processos:	2,04484364	204%
Resultado 6 Processos:	2,59406773	259%

PROGRAMAS MAIS EFICIENTES		
Speed UP PARALELO		
	Resultado	Percentagem %
Resultado 2 Processos:	1,85806605	186%
Resultado 4 Processos:	3,15837998	316%
Resultado 6 Processos:	3,39927045	340%

Como já constatado anteriormente, em ambos os programas a solução paralela é a que contém um maior aumento na eficiência e o SpeedUP é quase atingido no mais eficiente. No entanto, podemos afirmar que a solução paralela é a que terá maior redução de tempo.

Podemos afirmar que com 2 processos o tempo ganho é de quase 30%, enquanto no programa mais eficiente é de 85%. Na tentativa com 4 processos atingimos 104% de diminuição de tempo, face ao sequencial no programa menos eficiente e no mais eficiente de 216%. Na tentativa com 6 processos temos um aumento de 159% no menos eficiente e no mais eficiente o valor fica próximo dos 240%.

Conclusão

De acordo com os resultados, a execução paralela mostrou-se claramente uma melhoria significativa. Mas há certos aspetos a ter em conta.

As abordagens simultâneas nunca devem ser utilizadas para acelerar pequenos problemas de cálculo. A razão para isto é o tempo necessário para inicializar e finalizar bibliotecas, objetos de exclusão mútua (*mutex*), semáforos, entre outros, já que têm um grande impacto na produção.

Por outro lado, para um grande problema de computação, apesar de haver uma sobrecarga extra, o desempenho global aumenta. É necessário compreender a aplicabilidade, a conceção e a implementação adequada para desfrutar do desempenho.

Concorrência significa que um pedido está a progredir em mais do que uma tarefa ao mesmo tempo. Se o computador tiver apenas um CPU, a aplicação pode não progredir em mais do que uma tarefa ao mesmo tempo, mas mais do que uma tarefa está a ser processada, de forma sequencial, dentro da aplicação. A tarefa não termina por completo antes de começar a seguinte.

Concorrência significa executar múltiplas tarefas ao mesmo tempo, mas não necessariamente em simultâneo. Há duas tarefas executadas simultaneamente, mas estas são executadas num CPU de 1 *core*, pelo que o CPU decidirá executar primeiro uma tarefa e depois a outra, ou executar metade de uma tarefa e metade de outra, entre outras hipóteses. Duas tarefas podem ser iniciadas, executadas e concluídas em períodos de tempo sobrepostos, ou seja, a *task*-2 pode ser iniciada mesmo antes da *task*-1 ser concluída. Tudo depende da arquitetura do sistema.

Paralelismo significa que uma aplicação divide as suas tarefas em sub-tarefas mais pequenas que podem ser processadas em paralelo, como por exemplo, em vários CPU's ao mesmo tempo.

O paralelismo não requer a existência de duas tarefas. Ele executa literalmente partes de tarefas ou múltiplas tarefas, ao mesmo tempo que utiliza a infraestrutura *multi-core* do CPU, atribuindo um núcleo a cada tarefa ou sub-tarefa.

Numa analogia "paralela", e continuando com o mesmo exemplo, a regra continua a ser: cantar e comer em simultâneo, mas desta vez, joga-se numa equipa a dois. Provavelmente comerá e deixará a sua amiga cantar (porque ela canta melhor e você come melhor). Assim, sendo, as duas tarefas serão executadas em simultâneo, e chama-se a isso paralelismo.

O paralelismo requer um hardware com múltiplas unidades de processamento, essencialmente. Num CPU de um só *core*, pode obter-se simultaneidade, mas não paralelismo. O paralelismo é um tipo específico de processamento paralelo onde as tarefas são executadas em simultâneo.

Em sistemas simultâneos, múltiplas ações podem estar em curso, podendo não ser executadas ao mesmo tempo. Entretanto, ações múltiplas são executadas simultaneamente em sistemas paralelos. De facto, a simultaneidade e o paralelismo, são conceptualmente sobrepostos até certo ponto, mas "em curso" tornam-se claramente diferentes.

A concorrência consiste em lidar com muitas coisas ao mesmo tempo. O paralelismo tem a ver com fazer muitas coisas ao mesmo tempo.

Uma aplicação pode ser concorrente, mas não paralela, o que significa que processa mais do que uma tarefa ao mesmo tempo, não existindo duas tarefas a serem executadas ao mesmo tempo e no mesmo instante.

Uma aplicação pode ser paralela, e ao mesmo tempo não o ser, o que significa que processa múltiplas sub-tarefas de uma tarefa num CPU *multi-core* em simultâneo.

Uma aplicação não pode ser paralela, nem coexistente, o que significa que processa todas as tarefas, uma de cada vez, de forma sequencial.

Uma aplicação pode ser paralela e concorrente, o que significa que processa múltiplas tarefas ao mesmo tempo num CPU com vários *cores* em simultâneo.

Finalizando espero ter conseguido explicar as diferenças de ambos. Certamente haverá casos em que teremos mais vantagens em conduzir a aplicação para um paradigma mais concorrente e em outros mais paralelo, ou quiçá uma junção de ambos que também é possível. No entanto, e olhando para os nossos seis programas podemos afirmar categoricamente que a versão paralela de ambos será muito mais rápida e eficiente que a concorrente e sequencial isto deve-se assumidamente ao processamento em simultâneo e a divisão de recursos ser feita através do processo a *thread* que é a parte executável de um processo é criada à mesma mas em vez de estarem todas no mesmo processo estão divididas o que garantirá uma performance melhor e traduzindo-se em tempos de computação mais reduzidos evitando o overhead de *threads*. Podemos também verificar que apesar de no caso apresentado não ser realmente muito vantajoso a competição pelos recursos utilizada na versão concorrente poderá trazer alguma vantagem com a utilização continua de memória e de outros recursos da máquina.

"Se um único computador (processador) consegue resolver um problema em N segundos, podem N computadores (processadores) resolver o mesmo problema em 1 segundo?"