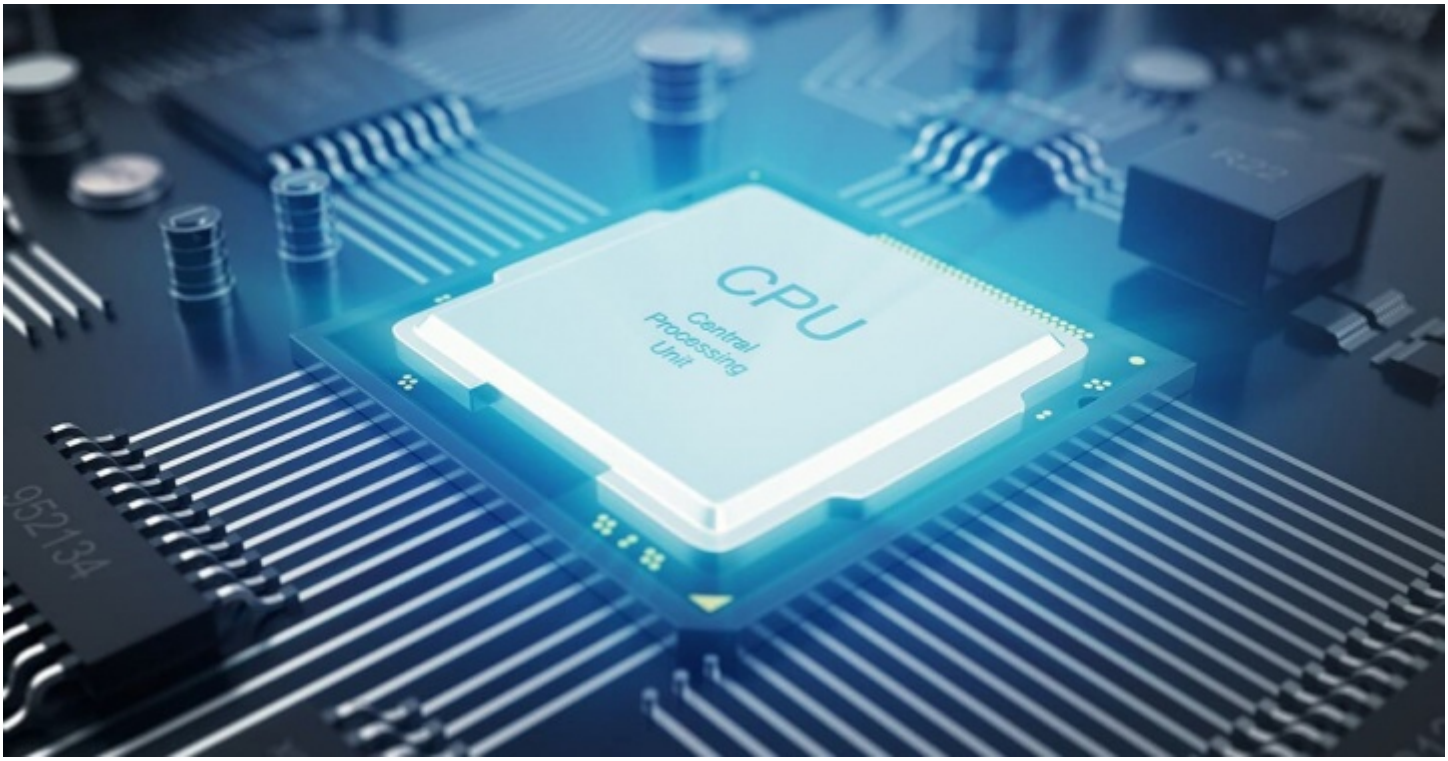
Sistemas Paralelos e Distribuídos

Relatório Técnico 1

Autores

Bruno Mendes

a62181



UAlg
UNIVERSIDADE DO ALGARVE

Universidade do Algarve

Faculdade de Ciências e Tecnologia

Engenharia Informática

18/2/2020

Resumo

O presente relatório descreve a implementação e análise de vários programas de cálculo de factores de um dado valor. Para este relatório foram implementadas as funções de forma a executar sequencialmente e em paralelo, onde serão recolhidados os tempos de execução e avaliado o desempenho demonstrando a importância do processsamento em paralelo.

Agradecimentos

Diogo Cordeiro - por me lembrar que a aceleração e eficiência existiam.
Leandro Quintans - pela discussão/sugestão de ideias, testes de velocidade do meu código e competição.

Índice

- 1. Introdução
  - 1.1. Objectivos
  - 1.2. Motivação
  - 1.3. Metedologia
  - 1.4. Estratégia
  - 1.5. Principais resultados/conclusões obtidos
  - 1.6. Estrutura Do Relatório
- 2. Enquadramento
  - 2.1. Conceitos
  - 2.2. Comandos
  - 2.3. Ambiente de Execução
    - 2.3.1. FCT-UALG
    - 2.3.2. *Personal Computer*
- 3. Desenvolvimento
  - 3.1. Descrição do Problema
  - 3.2. Implementação das Funções de Divisão de Factores
  - 3.3. Processamento Sequencial
    - 3.3.1. Alternativas de Desenho
    - 3.3.2. Implementações
    - 3.3.3. Recolha de dados
  - 3.4. Processamento Paralelo com *Pthreads*
    - 3.4.1. Descrição
    - 3.4.2. Alternativas de Desenho
    - 3.4.3. Implementações
    - 3.4.4. Recolha de Dados
  - 3.5. Processamento em Paralelo com *OpenMP*
    - 3.5.1. Alternativas de Desenho
    - 3.5.2. Implementações
    - 3.5.3. Resultados Obtidos
- 4. Análise de resultados e Discussão
  - 4.1. Análise
    - 4.1.1. Tempo de procura de factores em processamento paralelo vs sequencial
    - 4.1.2. Aceleração de *Pthreads* e *OpenMP*
    - 4.1.3. Eficiência de *Pthreads* e *OpenMP*
    - 4.1.4. Execução de *OpenMP* com N *threads* para *Longs*

- 5. Conclusão
- 6. Referências

1. Introdução

1.1. Objetivos

- Implementar uma ou várias funções sequencias e com paralelismo que descubram os factores de um dado número e ordenar os mesmos.
- Implementar e utilizar as regras para saber se 2, 3, 4, 5, 6, 9 e 10 sao factores.
- Comparar os resultados entre as funções sequenciais e funções com paralelismo, com *pthreads* e *OpenMP*.

1.2. Motivação

A motivação deste relatório é entender quando se deve usar processamento em paralelo e o quão vantajoso é este comparado com processamento sequencial. No nosso caso vamos procurar os factores de um número. Por exempo se uma pessoa tentar descobrir os factores sozinha, terá que dividir o número dado por todos os números até a raiz do dado número. Se o número for 121 será necessário dividir o número 11 vezes, mas se este processo for feito com a ajuda de alguém, pode-se separar as tarefas, uma pessoa divide de 1 até 6 e a outra faz de 7 até 11, assim há poupança de tempo. No caso dos computadores é semelhante só que tudo numa questão de milisegundos, e para tal este relatório foi feito para demonstrar que a programação paralela é vantajosa para situações em que a função não precise ser executada sequencialmente.

1.3. Metedologia

- **recolha de dados:** para a recolha de dados foi utilizado a função time para que nós pudessemos não só calcular o tempo de execução da função mas também o tempo efetivo de uso do cpu e o tempo de espera.

1.4. Estratégia

- **Não utilizar recursividade:** A recursividade requer que vários resultados sejam mantidos em memória física, tendo em conta que um dos objectivos é programar em paralelo de forma a que um programa execute mais rapidamente utilizando *threads*. No entanto, a utilização de mais

threads implica a utilização de mais memória, então para evitar a escassez de memória física, decidi não utilizar recursividade

- **Utilização de comandos e argumentos:** Permite que não seja necessário estar a comentar código para que outras funções executem, facilitando o teste de código e permite que várias combinações de inputs sejam possíveis através do uso da linha de comandos
- **Uso de testes unitários:** Permite a facilidade de teste de código e evitar erros
- **Uso de macros:** Facilidade de usar funções
- **Uso de *Strings*:** Facilidade em manipular cada elemento da *String* em realação aos *Integers*
- **Criar múltiplas implementações:** A criação de múltiplas implementações é importante porque permite que hajam mais testes e logo mais resultados para analisar
- **Utilizar *makefiles*:** para evitar reescrever o comando para compilar o código, criou-se um ficheiro makefile para facilitar o processo.
- **Utilizar múltiplos ficheiros de código:** Separação do código por múltiplos ficheiros de forma a facilitar a leitura do mesmo

1.5. Principais resultados/conclusões obtidos

Os principais resultados obtidos foram que a programação em paralelo é muito mais eficiente do que a sequencial e mesmo dentro da programação por *pthreads* e *OpenMP* existe uma grande diferença, sendo o *OpenMp* muito mais eficiente.

1.6. Estrutura Do Relatório

Os próximos capitulos vão enquandrar o leitor na linguagem ténica e mostrar como foi feito o desenvolvimento das funções e a recolha e análise das mesmas.

2. Enquadramento

2.1. Conceitos

- ***qsort*:** método de ordenação de um conjunto de elementos sobre uma regra definida com velocidade $O(nLog(n))$ (ver: *qsort*)
- ***insertion sort*:** método de ordenação que ordena um item de cada vez da lista, apesar da velocidade deste ser no pior dos casos de $O(n^2)$. No entanto se os elementos tiverem quase ordenados então o tempo de execução é quase $O(n)$
  (ver: *insertion sort*).
- **factores de um número:** número que divido por outro tem resto de 0, sendo o divido e o resultado dois dos seus factores
- **processo:** programa em execução.
- ***thread***: tarefa de um processo que pode ser executado em simultâneo.
- ***POSIX threads* (*p_threads*):** padrão *POSIX* para *threads*, o qual define a API das *threads* utilizadas neste programa.
- ***mutex* (mutual exclusion)** é um mecanismo de controlo de acesso a um recurso, permitindo que apenas seja acessido por um limite de *threads*
- **processamento sequencial:** processamento em que os comandos são executados de forma sequencial, ou seja, um após outro
- **processamento em paralelo:** processamento em que 2 ou mais comandos são executados em simulatâneo.
- **User CPU time:** "O tempo que o CPU gasta a executar, no espaço do utilizador, o processo que derivou do programa A"
- **System CPU time:** "O tempo que o CPU gasta a executar, no espaço do núcleo, o processo que derivou do programa A, i, é, a execução de rotinas do sistema opetativo desencadeadas".
- **Waiting time:** "tempo de espera até que uma operação de E/S tenha sido concluída ou devido à execução de outros processos".
- **Aceleração(S):**"é definida como a razão entre o tempo de execução dum problema num único processador t1 e o tempo necessário na resolução desse mesmo problema em p processadores idênticos, tp": (ver: aceleração)
  $S = \frac{t_1}{t_p}$ (1)
- **Eficiência:** "Define-se eficiência como sendo a fracção de tempo que osprocessadores realizam trabalho útil, ou seja, o quociente entre aceleração e o número de processadores", ou seja, a capacidade de aproveitamento dos recursos à diposição do processador.(ver: eficiência)
  $E = \frac{S}{p}$(2)

2.2. Comandos

```
#Versão da bash
$ ${BASH_VERSION}
# Versão do Sistema Linux
$ lsb_release -a
# Informação sobre o cpu
$ cat /proc/cpuinfo
# Número de threads
$ nprocs
# Nome do modelo do cpu
$ cat /proc/cpuinfo  | grep 'name'| uniq
# Compilar o programa
$ make
# Compilar todos os programas entre 10^1 até 10^10
$ ./script_get_time_of_functions.sh
# Compilar OpenMP para 4, 8, 16, 32, 64, 128 threads para input entre 10^11 e 10^18
$ ./script_run_open_mp_multiple_threads
# excutar o programa normalemente
$ ./main <args> <number> #ver lista de args em args.md
```

2.3. Ambiente de Execução

2.3.1. FCT-UALG

- **Linguagem de Programação:**C
- **Compilador:** compilador *gcc* com standard *gnu99*
- **Sistema Operativo:** Debian GNU/Linux 9.12 (stretch)
- **Versão da *BASH*:** 4.4.12(1)
- **Processador:** Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz, mais informação disponível aqui

2.3.2. *Personal Computer*

- **Linguagem de Programação:**C
- **Compilador:** compilador *gcc* com standard *gnu99*
- **Sistema Operativo:** WSL 1 Ubuntu 16.0.4.6 LTS
- **Versão da *BASH*:** 4.3.48(1)
- **Processador:** com o processador intel i7-7700k, mais informação disponível aqui

3. Desenvolvimento

3.1. Descrição do Problema

O problema em questão é descobrir os factores de um dado número usando a multiplicação o mínimo possível. Para tal foram usadas algumas regras conhecidas, como:

- **divisão por 2:** se o último digito de um número for par então o número é divisível por 2;
- **divisão por 3:** se a soma dos dígitos for um múltiplo de 3, então o número é divisível por 3, então se somar-mos os digitos da soma dos digitos até ficarmos apenas com um digito no final, então o resultado final vai ser um número entre 1 e 9, logo se esse resultado for 3, 6 ou 9, nós sabemos que é divisível por 3
- **divisão por 4:** se os últimos 2 números forem divisíveis por 4, então o número é divisível por 4
- **divisão por 5:** se o último número for 0 ou 10, então é divisível por 5
- **divisão por 6:** se for divisível por 2 e 3 então também é divisível por 6
- **divisão por 9:** semelhante ao 3, mas a soma dos digitos tem de ser igual a 9
- **divisão por 10:** se o último digito for 0 ou se for divisível por 2 e por 5, então é divisível por 10.

De resto foi só implementar um ciclo que executava as funções enquanto estas fossem menores ou iguais que a raíz do dado número e guardava os números no array.
Esse array depois é ordenado com o *insertion sort*.

3.2. Implementação das Funções de Divisão de Factores

- **divisão por 2:** para sabermos se um número é divisível por 2, então basta ver se o último dígito é par, logo como estamos a trabalhar com *strings* podemos ir buscar o último char da string e ver se pertence ao conjunto {0, 2, 4, 6, 8}
- **divisão por 3:** A divisão por 3 usou-se o método de converter cada elemento da *string* para *int* e somar esses valores criando uma nova *string* em loop até obter-mos um único valor, se esse valor pertencer a {3, 6, 9} então o número é divisível por 3.
- **divisão por 4:** semelhante à implementação do 2, foi necessário encontrar os 2 elementos e dividí-los por 4, se obtiver-mos resto de 0, então o número é divisor de 4.
- **divisão por 5:** também semelhante à implementação do 2, se o último algarismo da *string* estiver contido em {0, 5} então é divisível por 5
- **divisão por 6:** utilizando as funções já implementadas da divisão por 2 e 3, podemos saber se é divisível por 6.

- **divisão por 9:** reutilizando a função da divisão por 3, podemos verificar se o número obtido é 9, se for esse o caso, então é divisível por 9
- **divisão por 10:** utilizou-se 2 funções sendo que só é necessário utilizar uma, a primeira é semelhante à da implementação por 2, em que apenas precisamos de verificar se o último número é igual a 0, e a outra é reutilizando as funções de de divisão do 2 e do 5 previamente implementadas.

Para ordenadar foram implementado inicialmente 3 funções de sort, *qsort*, *mergesort* e *inserition sort*, no entanto utilizou-se apenas o *insertion sort* visto que o array final não estava desorganizado por isso o tempo de ordenação será semelhante a qualquer um dos mencionados anteriormente.

3.3. Processamento Sequencial

3.3.1. Alternativas de Desenho

Poderou -se utilizar o primeiro elemento do ARGV para decidir que tipo de função iria ser chamada, no entanto, para a mesma função nós iriamos querer imprimi-la com prints e sem prints, com tempo e sem tempo, com sort e sem sort, as vezes 2 destes elementos misturados, e utilizando so o primeiro elemento do argv, teriamos que implementar 9 diferentes alternativas para a utilização da mesma função, e se tivessemos 2 ja seriam 18, então, recorreu-se a utilização da função *getopt()* que também foi utilizada no guia 0 da cadeira.

3.3.2. Implementações

Foram feitas 2 implementações para a versão sequencial:

- A primeira implementação apenas executa todas as funções até à $\sqrt{(n)}$ sem ter em consideração aos resultados anteriores.
- A versão otimizada tem em consideração aos resultados obtidos anteriormente, que caso um número não seja divisível por 2 então não irá dividir por nenhum número par, se não for divisível por 3, então não irá dividir por 6 nem por 9, se não for divisível por 5, então não irá dividir por 9, isto faz com que o número de itereações seja inferior.

3.3.3. Recolha de dados

Ambas as implementações foram executadas 100 vezes para cada um dos números os $10En$ em que n é qualquer inteiro entre 0 e 10 com *insertion_sort*, anotando o *real time*, *user time* e também o *system time* usando uma *script* de *bash*, que correu o seguinte comando para os múltiplos números

```
#sequencial sem otimização
$ time ./a.out -l1 $((10**$i))#i é qualquer número inteiro entre 1 e 10
#sequencial com otimização
$ time ./a.out -l2 $((10**$i))
```

3.4. Processamento Paralelo com *Pthreads*

3.4.1. Descrição

O problema descrito é o mesmo descrito no capítulo de processamento squencial. Mas como agora temos várias *threads* podemos repartir o processo em várias partes, em que cada parte requere um trabalho idêntico, porque caso fique muito desequilibrado, não irá compensar em ter um processamento paralelo.

3.4.2. Alternativas de Desenho

Para colocar valores num array existem algumas maneiras de o fazer sem que as *threads* sobreponham os valores umas em cima das outras. Pode-se simplesmente usar uma formula matemática para calcular as posições dos arrays de forma a que nunca se sobreponham, pode-se alocar um array a 4 partes diferentes do array no entanto ficam muito dispersos, ou pode-se usar mutexes.

3.4.3. Implementações

Este problema foi implementado de 4 formas diferentes de forma a testar qual seria a melhor maneira de implementar threads para esta situação:

- **Primeira implementação:** Utilizou-se 2 *threads* em que uma faz as funções de 2, 3, 4, 5, 6, 9 e 10 e a outra faz o resto das operações, e colca -se os valores e aloca-se uma parte do array final para colocar estes resultados
- **Segunda implementação:** Utilizou-se 3 *threads* semelhante à situação anterior, só que em vez de uma *thread* fazer as divisões todas (excepto aquelas 7), temos uma *thread* para a divisão de impares e outra de pares, o número não for divisivel por 2, então a *thread* não irá executar
- **Terceira Implementação:** Esta implementação, à sugestão do aluno Leandro Quintans, dividir o array em N partes, dependendo da escolha do utilizador e executar N threads em que as threads vao fazer da posição $nThread * \frac{nInput}{nTotalThreads} + 1$ até $nThread * \frac{nInput}{nTotalThreads}$, caso o número seja impar, as *threads* ignoram os números pares, no entanto colocam a alocar os valores numa parte do array alocada às mesmas

- **Quarta Implementação:** Semelhante à anterior, só que são utilizados mutexes, logo os valores sao colocados de forma consecutiva

3.4.4. Recolha de Dados

Todas as implementações foram executadas 100 vezes para cada um dos números os $10En$ em que n é qualquer inteiro entre 1 e 10 com *insertion_sort*, anotando o *real time*, *user time* e também o *system time* usando uma *script* de *bash*, que correu o seguinte comando para os múltiplos números

```
#2 threads
$ time ./a.out -l3 $((10**$i))#i é qualquer número inteiro entre 1 e 10
#3 threads
$ time ./a.out -l4 $((10**$i))
#4 threads sem mutexes
$ time ./a.out -l5 $((10**$i))
#4 threads com mutexes
$ time ./a.out -l6 $((10**$i))
```

3.5. Processamento em Paralelo com *OpenMP*

3.5.1. Alternativas de Desenho

Ponderou-se em incialmente utilizar apenas o `# pragma omp parallel private (nthreads, tid)` onde se utilizaria o thread number para selecionar em que parte do *array* a *thread* iria atuar semelhante ao que acontece na implementação de N mutexes, no entanto `# pragma omp parallel for` já faz isso por definição. Outra hipotese foi utilizar `# pragma omp parallel for private(array)` onde cada *thread* tem um array local onde são colocados os factores encontrados pelas *threads*, no entanto, esta implementação não oferece quaisquer vantagens a `# pragma omp parallel for` logo foi excluida. Para controlar o acesso ao *array* de resultados inicialmente utilizou-se o `# pragma omp critical` mas como esta implementação o que faz é parar o código, ou seja, para todas as *threads* para colocar valores dentro da zona crítica então, escolheu-se utilizar os *mutexes* visto que estes não opõem qualquer tipo de paragem para as outras *threads*.

3.5.2. Implementações

A implementação de *OpenMP* foi bastante fácil, apenas reutilizou-se a versão sequencial otimizada e adicinou-se `# pragma parallel for` para que esta fosse paralelizada, depois foi só adicionar um mutex lock como nas *pthreads* para a região critica.

3.5.3. Resultados Obtidos

Todas as implementações foram executadas 100 vezes para cada um dos números os $10En$ em que n é qualquer inteiro entre 1 e 10 com *insertion_sort*, anotando o *real time*, *user time* e também o *system time* usando uma *script* de *bash*, que correu o seguinte comando para os múltiplos números

```
# OpenMP
$ time ./a.out -l7 $((10**$i))#i é qualquer número inteiro entre 1 e 10
```

4. Análise de resultados e Discussão

4.1. Análise

4.1.1. Tempo de procura de factores em processamento paralelo vs sequencial

Podemos observar que as funções de 2 e 3 *threads* têm tempo superior ao sequencial devido à forma de como foram desenhados demonstrando assim que esta implementação não tem sucesso. No entanto as *pthreads* já têm tempo melhor que o sequencial para números na ordem dos 10^9 sendo que a versão com mutexes é ligeramente mais rapida. Em relação ao *OpenMP*, o tempo de execução é muito mais baixo de qualquer outra implementação, tendo este uma aceleração de 3.7 para 10^{10} demonstrando assim em que quer sejam *pthreads* ou *OpenMP*, a arquitectura paralela é vatanjosa em relação à sequencial.

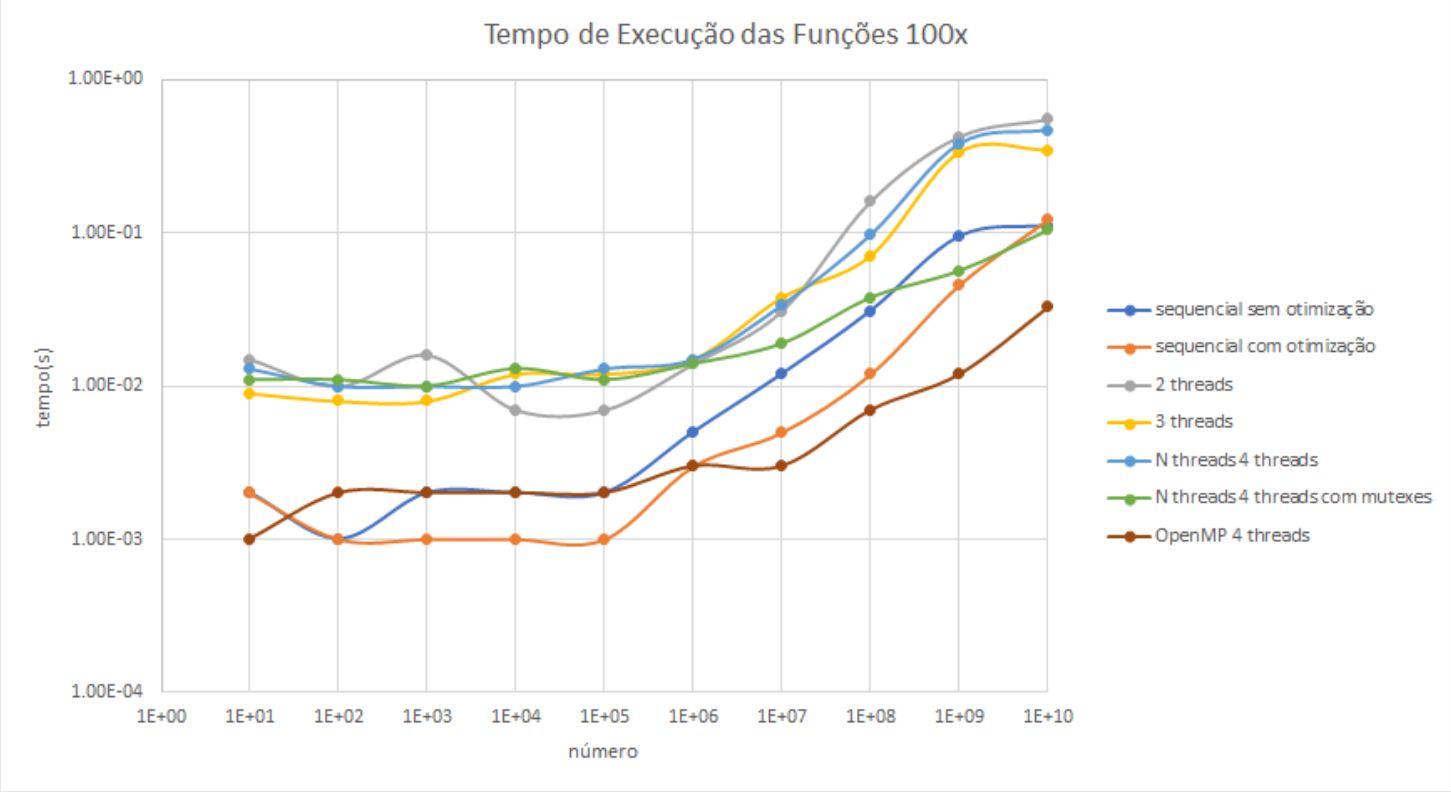


Gráfico 1 - tempo de execução das funções 100x

4.1.2. Aceleração de *Pthreads* e *OpenMP*

A utilização do *OpenMP* começa a a ser vatanjosa para números superiores a 10^6 enquanto com *pthreads* só se vê uma melhoria a partir de 10^{10}.

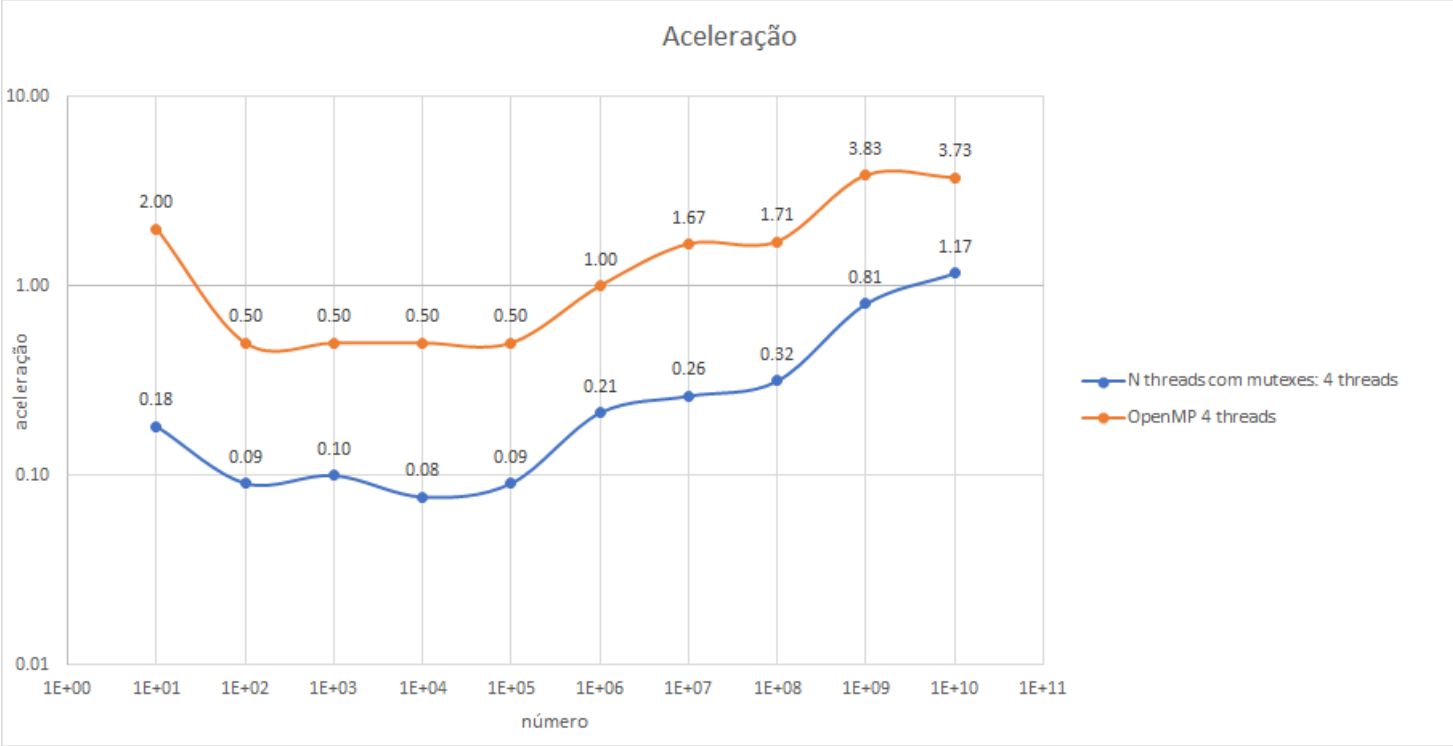


Gráfico 2 - Aceleração de Pthreads com mutexes e OpenMP

4.1.3. Eficiência de *Pthreads* e *OpenMP*

Como a aceleração de *OpenMP* já era mais rápida que a de *pthreads* e como ambas estão executar com 4 *threads* é de esperar que o *OpenMp* continue a ser mais rápido. Para comparação, a eficiência do *OpenMP* para 10^{10} é de 93% enquanto para as *pthreads* apenas de 29.29%, ou seja, *OpenMP* neste ponto é 3.17 vezes mais eficiente que a implementação com *pthreads*.

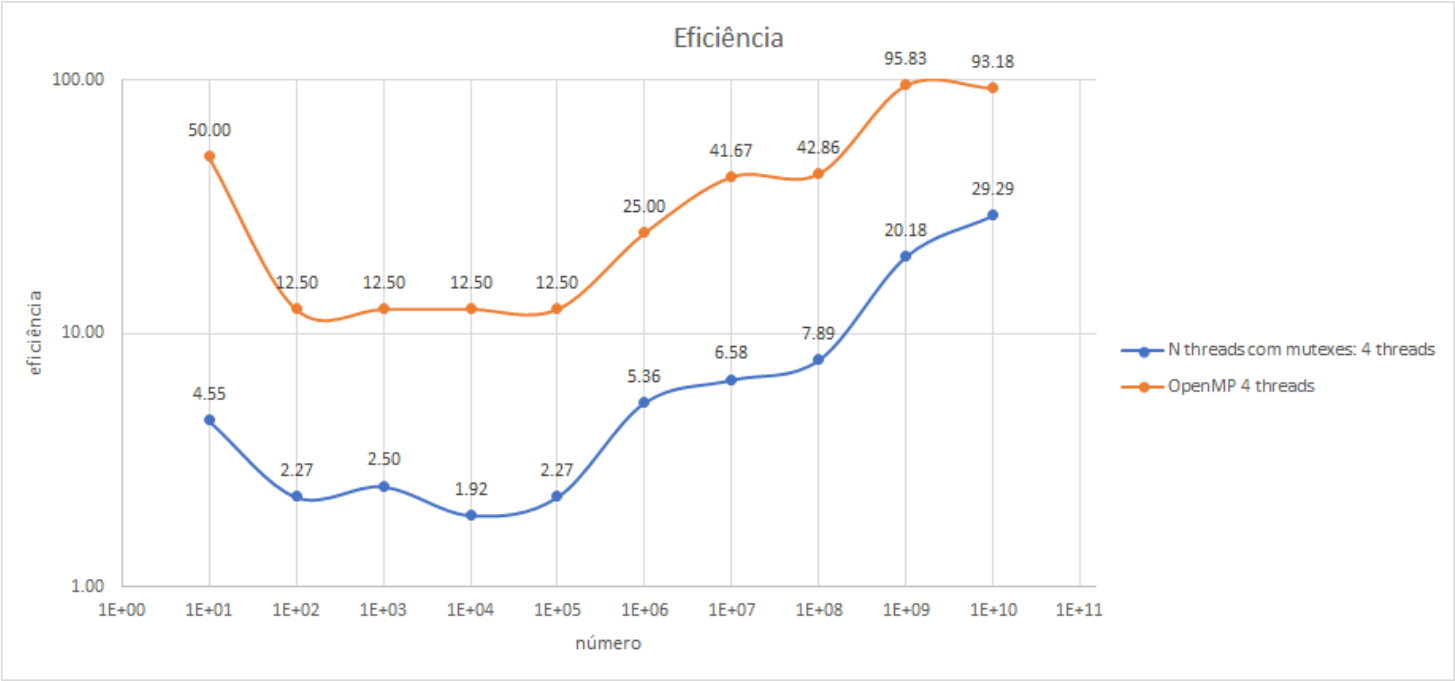


Gráfico 4 - Eficiência de Pthreads com mutexes e OpenMP

4.1.4. Execução de *OpenMP* com N *threads* para *Longs*

Observando os resultados obtidos podemos concluir que não há qualquer vantagem em relação à utilização de mais do que 4 threads para o processador do servidor da FCT-UALG

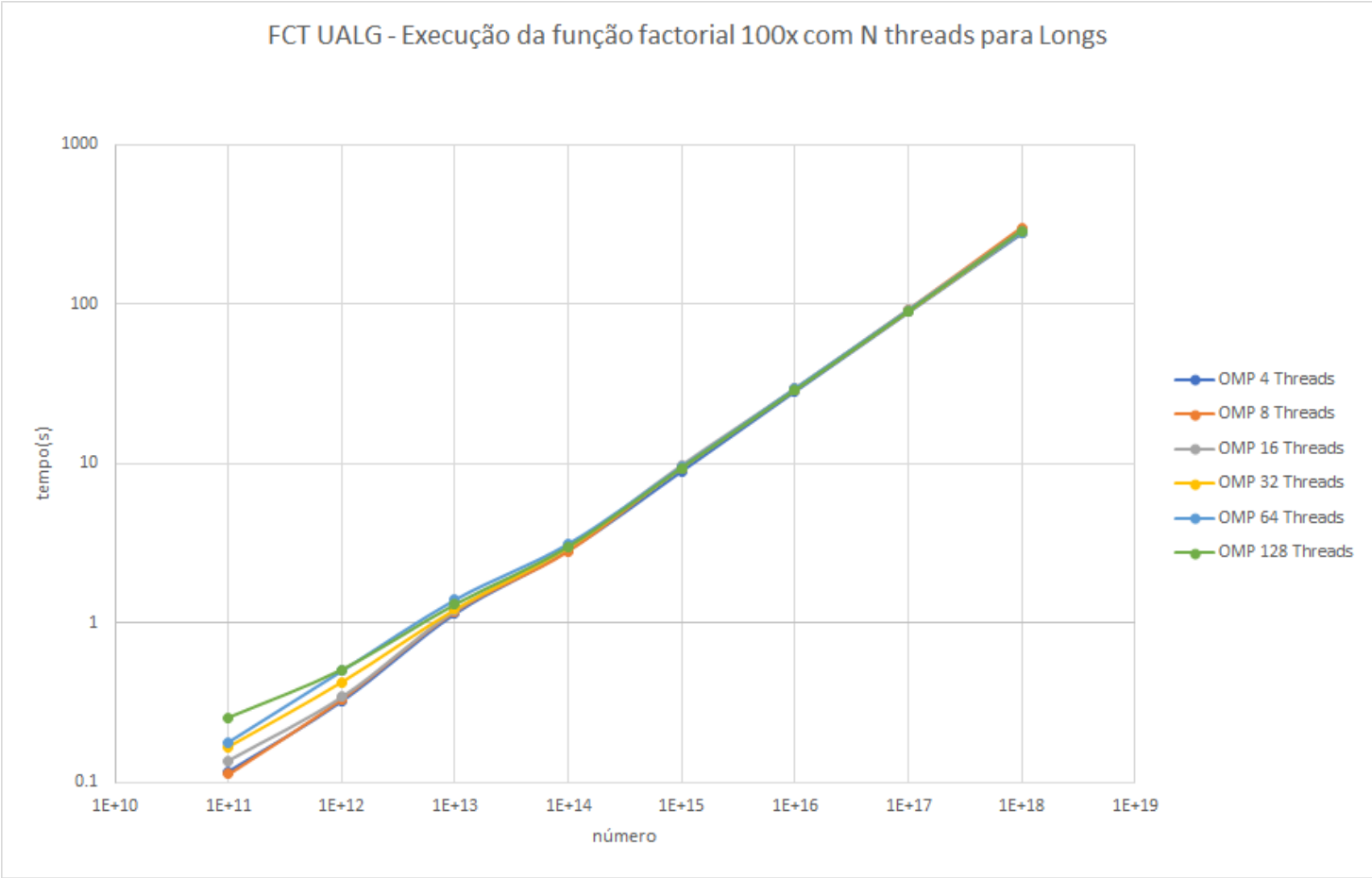


Gráfico 4 - número de threads vs velocidade de execução de longs no servidor fct-ualg

Observando os resultados obtidos podemos concluir que o uso de mais *threads* de facto afecta a velocidade de execução, no entanto a velocidade de execução não melhora ao ponto em que compense utilizar mais do que 8 threads

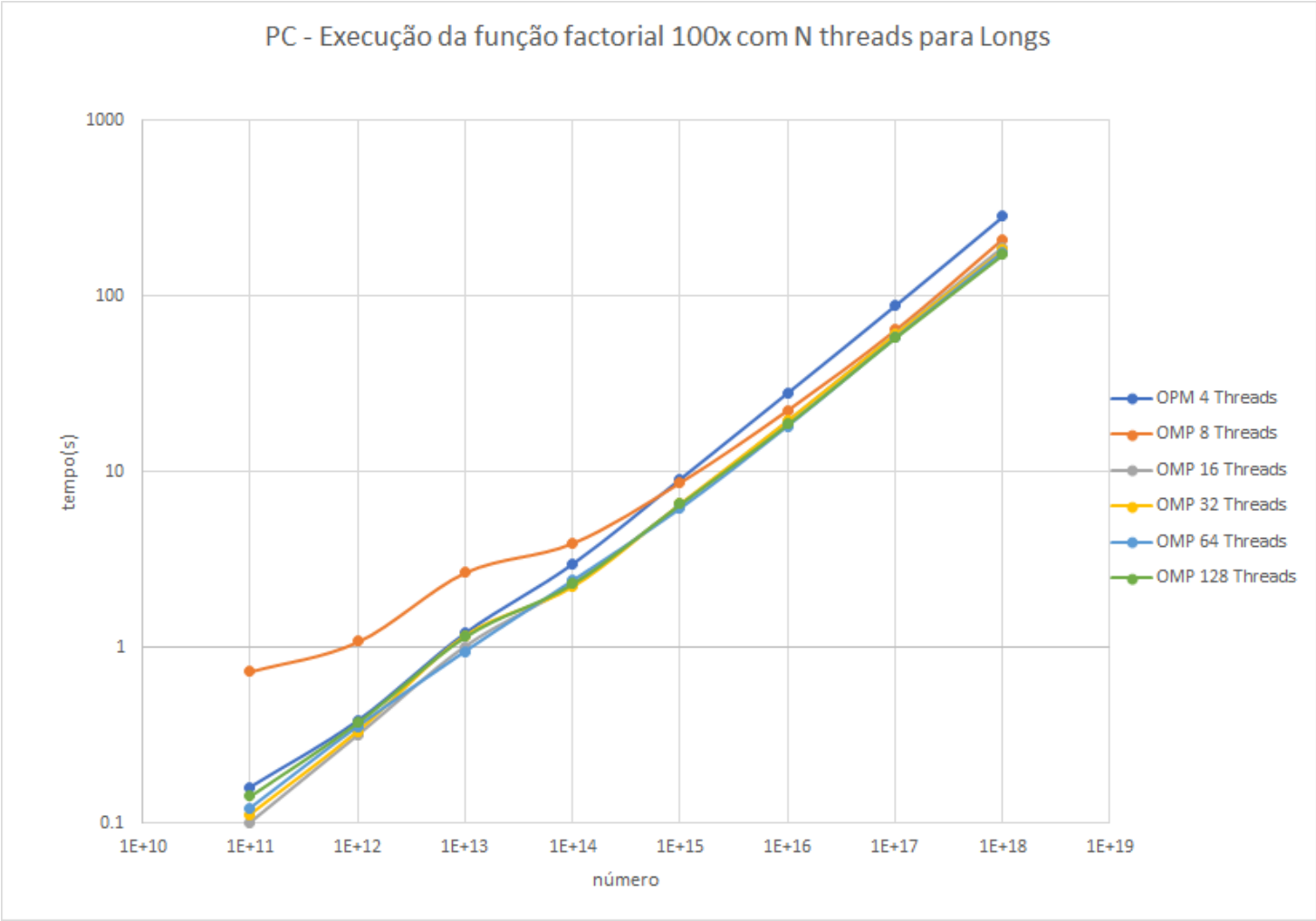


Gráfico 5 - número de threads vs velocidade de execução de longs no meu pc

5. Conclusão

O processamento em paralelo é de facto mais eficiente que o processamento sequencial quanto maior for o número de comandos do processo. No entanto para que possamos aplicar o processamento em paralelo temos que entender a natureza do problema e identificar de quais são os recursos que só podem ser acedidos por uma *thread*. Com o uso de *OpenMP* a implementação foi bastante simples e a velocidade comparada não só com o sequencial mas com outro estilo de programação paralela como *pthreads* é muito mais eficiente. No entanto, como observamos a partir de um certo ponto não compensa utilizar mais *threads*, então terá que se encontrar outro método de otimizar o tempo. Esse outro método será utilizar o método híbrido de programação distribuida com programação paralela, em que distribuimos o trabalho por vários computadores e cada computador faz processamento paralelo, que é o método mais comum de hoje em dia. A implementação deste método será utilizado no próximo trabalho com a implementação de MPI em parceria com *OpenMP*.

6. Referências

[1] "Utilizar argumentos na linha de comandos" - Disponível em: (https://tutoria.ualg.pt/2019/pluginfile.php/132031/mod_resource/content/1/00-Revisões.html)
[2] "Utilizar multiplos argumentos na linha de comandos" - Disponível em (https://www.gnu.org/software/libc/manual/html_node/Getopt-Long-Option-Example.html)
[3] "Como criar um foreach em C" - Disponível em (https://en.wikipedia.org/wiki/Foreach_loop#C)
[4] Tomblin Paul, "Como converter um *char* para *int* em C - Disponível em (https://stackoverflow.com/questions/868496/how-to-convert-char-to-integer-in-c)
[5] Swedgin, "Como criar um array de funções" - Disponível em (https://stackoverflow.com/questions/5309859/how-to-define-an-array-of-functions-in-c/35756809)
[6] Madeira e Moura, Maria Margarida - "Função de cálculo de tempo", exericio 5 - Disponível em (https://tutoria.ualg.pt/2019/pluginfile.php/132032/mod_resource/content/0/Guia1-Exercícios Iniciais.pdf)
[7] "Implementar MergeSort em C" - Disponível em (https://www.geeksforgeeks.org/merge-sort/)
[8] Guerreiro, Pedro - "Clonar Strings" - Disponível em (https://tutoria.ualg.pt/2019/pluginfile.php/116217/mod_resource/content/12/our_strings.c)
[9] Spikatrix - "Converter String para Long" - Disponível em (https://stackoverflow.com/questions/7021725/how-to-convert-a-string-to-integer-in-c)
[10] "Insertion Sort em C" - Disponível em (https://www.geeksforgeeks.org/insertion-sort/)
[11] "Sumatório de um *array* com *pthreads*" - Disponível em (https://www.geeksforgeeks.org/sum-array-using-pthreads/)
[12] "Exemplos de criação e utilização de *pthreads*" - Disponível em (https://computing.llnl.gov/tutorials/pthreads/?fbclid=IwAR1EzFsCa-cjQ_eluOzqvMF-agu726Wcs3rKAXqQyLp-oti7nrWuBnYXgmw#PthreadsAPI)
[13] "Calculadora de fatores" - Disponível em(https://www.calculatorsoup.com/calculators/math/factors.php)
[14] Khan Academy - "Como calcular factores" - Disponível em (https://www.khanacademy.org/math/pre-algebra/pre-algebra-factors-multiples)
[15] "Como escrever um relatório técnico" - Disponível em (http://w3.ualg.pt/~jmartins/tecnicascomunicacao/Como.escrever.um.relatório.pdf)
[16] "Imagem de um cpu" - Disponível em (https://pplware.sapo.pt/software/intel-software-overclock/)
[17] "Processamento em paralelo de *pthreads*" - Disponível em(https://tutoria.ualg.pt/2019/pluginfile.php/137807/mod_resource/content/0/3.Processamento paralelo.pdf)
[18] "API" - Disponível em (https://pt.wikipedia.org/wiki/Interface_de_programação_de_aplicações)
[19] "POSIX" - Disponível em (https://pt.wikipedia.org/wiki/POSIX)

[20] "Gerar documentação em C" - Disponível em (http://www.doxygen.nl/index.html)
[21] "Lista de funções em Markdown" - Disponível em (https://csrgxtu.github.io/2015/03/20/Writing-Mathematic-Fomulars-in-Markdown/)
[22] "Implementação de *qsort*" - Disponível em (https://en.wikipedia.org/wiki/Qsort)
[23] Madeira e Moura, Maria Margarida - "Pdfs de Avaliação de Performance" - Disponível em (https://tutoria.ualg.pt/2019/pluginfile.php/133291/mod_resource/content/1/2.performance_eval.pdf)
[24] Madeira e Moura, Maria Margarida - "Definição de eficiência e aceleração" - Disponível em (https://tutoria.ualg.pt/2019/pluginfile.php/133291/mod_resource/content/1/2.performance_eval.pdf)