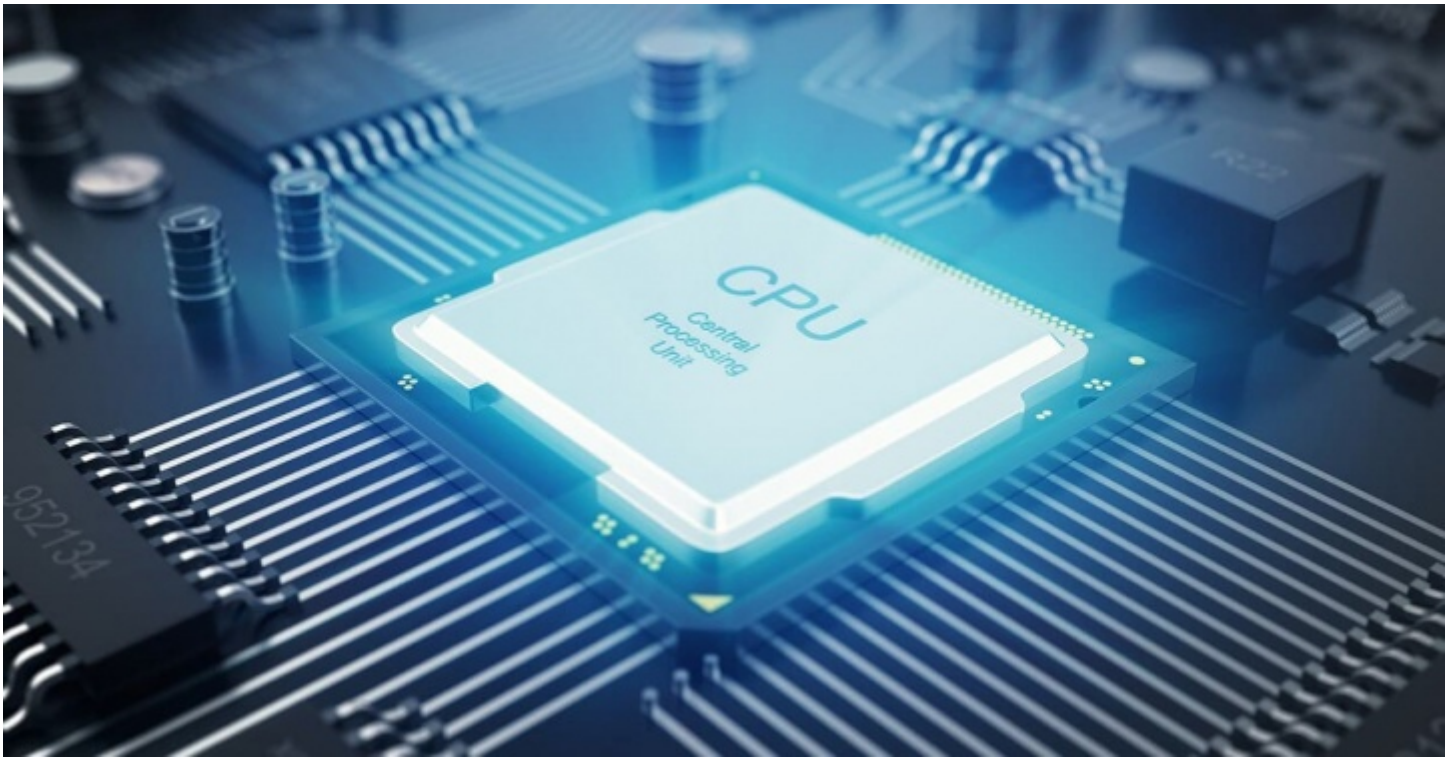
Sistemas Paralelos e Distribuídos

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 em processamento paralelo e processamento sequencial para a cadeira de Sistemas Paralelos e Distribuidos.
Para este vamos implementar funções que são processadas sequencialmente e outras em paralelo onde iremos recolher dados para múltiplos números e comparar os resultados obtidos e entender a importância da programação em paralelo.
No capítulo 5 poderemos observar que a programação paralela de facto usa mais poder de processamento no entanto o seu tempo total de execução é inferior porque estamos a dividir o problema em tarefas que irão ser executadas em simultâneo resolvendo o problemo mais rapidamente.

Agradecimentos

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

Indice

1. Introdução

1.1. Objectivos

- Implementar uma ou várias funções sequencias e com paralelismo que descubram a os factores de um dado número](https://www.khanacademy.org/math/pre-algebra/pre-algebra-factors-multiples) 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

1.2. Motivação

A motivação deste relatório é entender quando se deve usar processamento em paralelo e o quão vantajoso este é comparado com processamento sequencial. No nosso caso vamos procurar os factores de um número, se eu tentar descobrir os factores sozinho, terei que dividir o número dado por todos os números até a raiz do dado número. Se o número for 121 só preciso de dividir 121 onze vezes, mas se tiver um amigo comigo, eu posso dividir de 1 até 6 e ele faz de 7 até 11, assim já poupamos tempo. No caso dos computadores é semelhante só que tudo numa questão de milisegundos, e para tal este relatório foi feito para demonostrar que a programação paralela é vantajosa para situações em que a função não precise ser executada por ordem.

1.3. Metodologia seguida

- **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:** 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

1.4. Principais resultados/conclusões obtidos

Os principais resultados obtidos foram que as funções pthreads serão mais eficientes quanto maior for o número de operações necessárias, também se notou que apesar de o tempo gasto no cpu por parte das funções paralelas ser maior o seu tempo de execução real é bastante inferior porque a função é dividida em múltiplos bocados e esses bocados são executados em paralelo

1.5. 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

- ***Integer*:** número que pode ser escrito sem parte fracionaria (ver: *Integer*)
- ***String*:** cadeia de caracteres que representa uma palavra (ver: *String*)
- ***qsort*:** método de ordenação de um conjunto de elementos sobre uma regra definida com velocidade $O(nLog(n))$ (ver: *qsort*)
- **factores de um número:** número que divido por outro tem resto de 0, sendo o divido e o resultado dois dos seus factores
- **processo:** instância de um programa que pode estar a ser executado por 1 ou mais *threads*
- ***thread***: é um conjunto de instruções de um processo
- ***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, evitando que este seja alterado simultâneamente por várias *threads*
- **processamento sequencial:** processamento em que os comandos são executados de forma "linear", 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:** é a 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: $S = \frac{t_1}{t_p}$ (1) (ver: aceleração)
- **Eficiência:** Define-se eficiência como sendo a fracção de tempo que os processadores realizam trabalho útil, ou seja, o quociente entre aceleração e o número de processadores: $E = \frac{S}{p}$(2) (ver: eficiência)

2.2. Ambiente de Execução

Os programas foram escritos em C, executados no WSL com o processador intel i7-7700k

3. Execução com processamento Sequencial

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. Para a divisão por 2, para is buscar o ultimo digito

A implementação destas funções tem como o objetivo de evitar fazer cálculos desnecessários, no entanto para acedermos ao ultimo digito de um elemento em C temos que dividir o elemento e isso era o que queriamos evitar em primeiro lugar, para tal vamos converter o número para *String* assim é facil aceder ao último digito e apenas temos que converter esse para *Integer* e assim só teremos que ver se o último elemento é um 0, 2, 4, 6 ou 8.
Para o 3 teve-se que fazer uma soma dos elementos, mas como estamos a usar *Strings* foi facil de somar digito a digito, até chegarmos ao ponto em que só temos um digito.
Para o quatro não houve solução simples, porque teve-se que recorrer à divisão, mas como foi só à divisão dos últimos 2 digitos em que não é necessário quase processamento nenhum.
O 5, 6 e 10 foram baseados nas implementações do 2 e a do 9 na do 3.
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 qSort.

3.2. Alternativas de Desenho

Para construir a função do 3, ponderou-se inicialmente em fazer recursivamentea função. No entanto as funções recursivas requerem que seja guardada memória temporaria, logo se tivessemos multiplas *threads* a fazer cálculo e soma de digitos até ficarmos só com 1, haveria uma grande quantidade de memória que estaria a ser ocupada injustificadamente, esta tambem é a razão pela qual não se ponderou a utilização do mergesort.
Tambem se poderou 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 estava a ser utilizada no guia 0 da cadeira.

3.3. 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 considerção 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 comandos executados seja inferior.

3.4. Recolha de dados

Ambas as implementações foram executadas 100 vezes para cada um dos números 120, 5231, 52992, 999999, 2352992, 78954358, 514879867, 1318231997 e 2147483646 com qsort e sem *qsort* para saber também qual era o peso que a ordenação tem no tempo de execução total, e registou-se a média de tempos de cada situação.
Também se recolheram dados utilizando o comando seguido em baix para recolha do tempo do user CPU vs sytem CPU

```
$ time ./a.out -comandos <numero>
```

4. Execução com processamento em paralelo

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.

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.

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

4.4. Recolha de dados

Todas as implementações foram executadas 100 vezes para cada um dos números 120, 5231, 52992, 999999, 2352992, 78954358, 514879867, 1318231997 e 2147483646 com qsort e sem *qsort* para saber também qual era o peso que a ordenação tem no tempo de execução total, e registou-se a média de tempos de cada situação.
Também se recolheram dados utilizando o comando seguido em baix para recolha do tempo do user CPU vs sytem CPU

```
$ time ./a.out -comandos <numero>
```

5. Análise de resultados e Discussão

5.1. Resultados Obtidos

Tabela 1 - resultados obtidos da execução da função sequencial sem otimização

input	média do tempo de execução dos fatores(s)	média do tempo de execução da função de ordenação	média do tempo total de execução(s)	Diferença entre o tempo inicial(%)
120	0.000002	0	0.000002	0
5231	0.000004	0	0.000004	100
52992	0.000011	0.000003	0.000014	600
999999	0.000035	0.000001	0.000036	1700
2352992	0.000048	0	0.000048	15800
79854358	0.000276	0.000042	0.000318	40350
514879867	0.000785	0.000024	0.000809	69500
1318231997	0.001319	7.3E-05	0.001392	69500
2147483646	0.001724	6E-06	0.00173	86400

Tabela 2 - resultados obtidos da execução da função sequencial com otimização

input	média do tempo de execução dos fatores(s)	média do tempo de execução da função de ordenação(s)	média do tempo total de execução(s)	Diferença entre o tempo inicial(%)
120	0.000002	0.000002	0.000004	0
5231	0.000002	0	0.000002	-50

input	média do tempo de execução dos fatores(s)	média do tempo de execução da função de ordenação(s)	média do tempo total de execução(s)	Diferença entre o tempo inicial(%)
52992	0.000008	0.000001	0.000009	125
999999	0.000012	0	0.000012	200
2352992	0.000033	1E-06	0.000034	750
79854358	0.000203	0.000059	0.000262	6450
514879867	0.000278	0.000016	0.000294	7250
1318231997	0.000488	8E-06	0.000496	123000
2147483646	0.001307	0.000033	0.00134	33400

Tabela 3 - resultados obtidos da execução da função com 2 threads

input	média do tempo de execução dos fatores(s)	média do tempo de execução da função de ordenação(s)	média do tempo total de execução(s)	Diferença entre o tempo inicial(%)
120	0.000103	0	0.000103	0
5231	0.000115	2E-06	0.000117	13.59223
52992	0.000121	2E-06	0.000123	19.41748
999999	0.000154	0.000059	0.000273	76.69903
2352992	0.000214	1E-06	0.000034	165.0485
79854358	0.000645	0.000227	0.000872	74.6019
514879867	0.000806	0.000823	0.001629	1481.553
1318231997	0.001132	0.001548	0.00268	2501.942
2147483646	0.002734	0.001968	0.004702	4465.046

Tabela 4 - resultados obtidos da execução da função com 3 threads

input	média do tempo de execução dos fatores(s)	média do tempo de execução da função de ordenação(s)	média do tempo total de execução(s)	Diferença entre o tempo inicial(%)
120	0.00012	8E-06	0.000128	0
5231	0.000137	0.000018	0.000155	21.09375
52992	0.000123	0.000028	0.000151	17.96875
999999	0.000174	0.000038	0.000212	65.625
2352992	0.00018	0.000048	0.000228	78.125
79854358	0.000789	0.000802	0.000872	460.9375
514879867	0.000806	0.000823	0.001591	1142.969
1318231997	0.00118	0.00156	0.00274	2040.625
2147483646	0.001597	0.002424	0.004021	3041.406

Tabela 5 - resultados obtidos da execução da função genérica a executar com 4 threads

input	média do tempo de execução dos fatores(s)	média do tempo de execução da função de ordenação(s)	média do tempo total de execução(s)	Diferença entre o tempo inicial(%)
120	0.000141	2E-06	0.000143	0
5231	0.000178	0	0.000178	24.47552
52992	0.000172	0.000013	0.000185	29.37063
999999	0.000192	0.00002	0.000212	48.25175
2352992	0.000253	5E-06	0.000258	80.41958
79854358	0.000284	0.000313	0.000597	317.4825
514879867	0.000343	0.001013	0.001356	848.2517

input	média do tempo de execução dos fatores(s)	média do tempo de execução da função de ordenação(s)	média do tempo total de execução(s)	Diferença entre o tempo inicial(%)
1318231997	0.000473	0.001091	0.001564	993.7063
2147483646	0.000891	0.001936	0.002827	1876.923

Tabela 6 - resultados obtidos da execução da função genérica a executar com 8 threads

input	média do tempo de execução dos fatores(s)	média do tempo de execução da função de ordenação(s)	média do tempo total de execução(s)	Diferença entre o tempo inicial(%)
120	0.000538	0.000019	0.000557	0
5231	0.000552	5E-06	0.000557	0
52992	0.000549	0.00003	0.000579	3.949731
999999	0.000567	0.000016	0.000583	4.667864
2352992	0.00058	0.000024	0.000604	8.438061
79854358	0.000515	0.000337	0.000852	52.9623
514879867	0.000574	0.0008	0.001374	146.6786
1318231997	0.000793	0.001197	0.00199	257.2711
2147483646	0.000908	0.001697	0.002605	367.684

Tabela 7 - resultados obtidos da execução da função genérica a executar com 4 threads e mutexes

input	média do tempo de execução dos fatores(s)	média do tempo de execução da função de ordenação(s)	média do tempo total de execução(s)	Diferença entre o tempo inicial(%)

input	média do tempo de execução dos fatores(s)	média do tempo de execução da função de ordenação(s)	média do tempo total de execução(s)	Diferença entre o tempo inicial(%)
120	0.000154	0.000008	0.000162	0
5231	0.000153	0.000009	0.000162	0
52992	0.000177	7E-06	0.000184	13.58025
999999	0.000179	3E-06	0.000182	12.34568
2352992	0.000218	3E-06	0.000221	36.41975
79854358	0.000342	0.000016	0.000358	120.9877
514879867	0.000398	0.000013	0.000411	153.7037
1318231997	0.000466	0.000015	0.000481	196.9136
2147483646	0.000857	0.000018	0.000875	440.1235

Tabela 8 - resultados obtidos da execução da função genérica a executar com 8 threads e mutexes

input	média do tempo de execução dos fatores(s)	média do tempo de execução da função de ordenação(s)	média do tempo total de execução(s)	Diferença entre o tempo inicial(%)
120	0.000533	0.00003	0.000563	0
5231	0.000547	0.000027	0.000574	1.953819
52992	0.000496	0.000079	0.000575	21.31439
999999	0.000602	2E-06	00.000604	7.282416
2352992	0.000497	1E-05	0.000507	-9.94671
79854358	0.00061	8E-06	0.000618	9.769094
514879867	0.000528	5E-05	0.000578	2.664298

input	média do tempo de execução dos fatores(s)	média do tempo de execução da função de ordenação(s)	média do tempo total de execução(s)	Diferença entre o tempo inicial(%)
1318231997	0.000603	4.2E-05	0.000645	14.56483
2147483646	0.000927	0.000009	0.000936	66.2522

Tabela 9 - tempo de resposta para as várias funções com input 2147483646 e sem ordenação

função	user CPU time	system CPU time	waiting time
sequencial sem otimização	0.164	0.141	0.031
sequencial com otimização	0.126	0.125	0
2 pthreads	0.288	0.266	0.016
3 pthreads	0.155	0.234	0.016
N pthreads sem mutexes: 4 threads	0.092	0.219	0
N pthreads sem mutexes: 8 threads	0.1	0.234	0.063
N pthreads com mutexes: 4 threads	0.095	0.203	0.031
N pthreads com mutexes: 8 threads	0.097	0.281	0.031

Tabela 10 - tempo de resposta para as várias funções com input 2147483646 e com ordenação

função	user CPU time	system CPU time	waiting time
sequencial sem otimização	0.171	0.156	0.016
sequencial com otimização	0.136	0.141	0
2 pthreads	0.419	0.359	0.047
3 pthreads	0.329	0.422	0.016
N pthreads sem mutexes: 4 threads	0.263	0.234	0.031
N pthreads sem mutexes: 8 threads	0.1	0.234	0.063

função	user CPU time	system CPU time	waiting time
N pthreads com mutexes: 4 threads	0.095	0.25	0.016
N pthreads com mutexes: 8 threads	0.098	0.297	0.078

Tabela 11 - Aceleração das funções de N threads com mutexes em relação à função sequencial com otimização

Aceleração	4 threads	8 threads
120	0.014184397	0.003752345
5231	0.011235955	0.003656307
52992	0.046511628	0.016129032
999999	0.0625	0.019933555
2352992	0.130434783	0.06639839
79854358	0.714788732	0.332786885
514879867	0.810495627	0.526515152
1318231997	1.031712474	0.809286899
2147483646	1.466891134	1.409924488

Tabela 12 - Eficiência das funções de N threads com mutexes de 4 e 8 threads

Eficiência	4 threads	8 threads
120	0.354609929	0.046904315
5231	0.280898876	0.045703839
52992	1.162790698	0.201612903
999999	1.5625	0.249169435
2352992	3.260869565	0.829979879
79854358	17.86971831	4.159836066

Eficiência	4 threads	8 threads
514879867	20.26239067	6.581439394
1318231997	25.79281184	10.11608624
2147483646	36.67227834	17.62405609

5.2. Análise

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

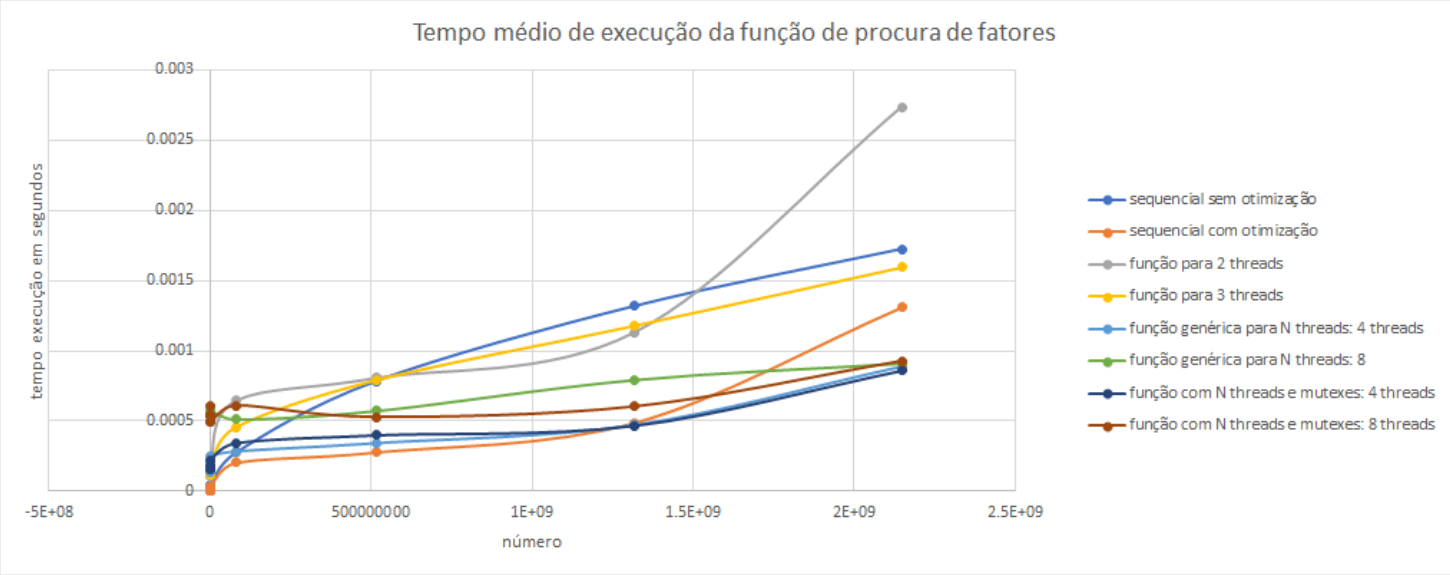


Gráfico 1 - comparação do tempo de execução da função factorial

Observando os dados obtidos no gráfico 1, podemos induzir que:

- O processamento sequencial é de mais eficiente para valores que precisam de poucas divisões porque para inicilizar uma thread tem algum custo de tempo e se o valor dado for baixo então o processamento sequencial termina os cálculos antes das *p_threads* serem usadas efetivamente
- Ainda no processamento sequencial podemos observar que a função otimizada executa sempre mais rapidamente e em algumas situações quase três vezes mais rapidamente (ver tabela 1)
- A função de 2 threads tem um tempo de execução elevado. Isso deve-se ao facto da forma como foi implementada, o trabalho distribuido pelas 2 *threads* está muito desequilibrado. Uma destas descobre apenas os factores de um dado número para 2, 3, 4, 5, 6, 9 e 10, enquanto a outra faz dos outros todos, para casos em que o número seja por exemplo 2147483646, esta thread terá

um tempo de utilização muito menor que a outra, esta descobre 7 factores enquanto a outra irá ter que descobrir 2147483639 factores. Esta implementação não cumpre o objetivo de demonostrar que para valores elevados o processamento em paralelo é de facto mais eficiente

- A função de 3 threads apenas "vence" a função sequencial sem otimização, isto é de esperar devido a problemas semelhantes com a implementação da mesma. O trabalho distribuido pelas *threads* é muito desequilibrado. Esta implementação tem o mesmo problema que a implentação da função de 2 *threads* , uma das *threads* também está a ser utilizada para executar as funções de 2, 3, 4, 5, 6, 9 e 10, o que para valores elevados, não faz uma diferença significativa. O outro problema é que esta função, como mencionado, tem uma thread para resolver os números pares e a outra os impares, no entanto se o número não for divisível por 2, então a *thread* pára de executar. Se o número for um ímpar elevado então vamos ter apenas uma *thread* a executar o que não é vantajoso a uma função sequencial, tendo em consideração estes pontos também podemos afirmar que esta implementação falha nos objetivos definidos
- Como podemos observar as funções genérica para N *threads* (4 e 8 neste caso) tanto as com e sem mutexes, são as mais rápidas a executar para números que poderão ter uma grande quantidade de factores. Isto deve-se ao facto de esta implementação não ter *threads* dedicadas para pares / impares e as funções das 7 funções indicadas, então teremos sempre N *threads* ativas a procurar factores o que faz com que esta implementação seja mais eficiente para a procura dos valores

5.2.2. Tempo de ordenação dos resultados das funções de processamento paralelo e sequencial

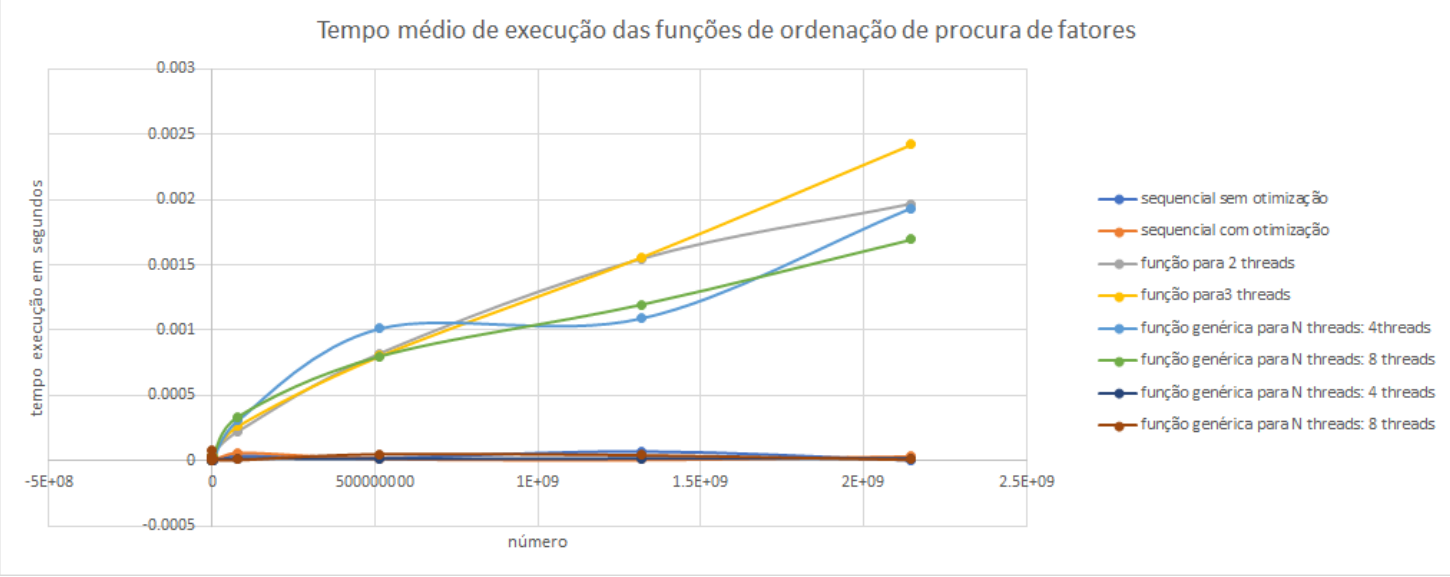


Gráfico 2 - comparação do tempo de ordenação nas diferentes funções de procura de factores

Observando os dados obtidos no gráfico 2, podemos induzir que:

- A ordenação a funções sequenciais é bastante mais rápida porque sabemos até que posição do array foi preenchida com valores, logo só precisamos ordenar até a posição do ultimo valor inserido
- A ordenação a funções em paralelo (a implementação de 2 threads, 3 threads, e genérica sem mutexes) não é tão linear porque os valores são colocados na parte do array que foi alocada à *thread*, logo os valores estão dispersos pelo array o que leva que o número de posições do array que precisa de ser ordenado seja superior ao número de factores encontrados, e como quanto maior for o número dado maior será o espaço alocado do array e por sua vez o tempo de ordenação também será maior.
- A ordenação a funções com processamento com mutexes, já temos controlo sobre o acesso ao array, logo podemos colocar todos os valores no array de forma a que saibamos até que ponto podemos fazer a ordenação, fazendo com que esta seja quase instântanea

5.2.3. Tempo total de execução das funções

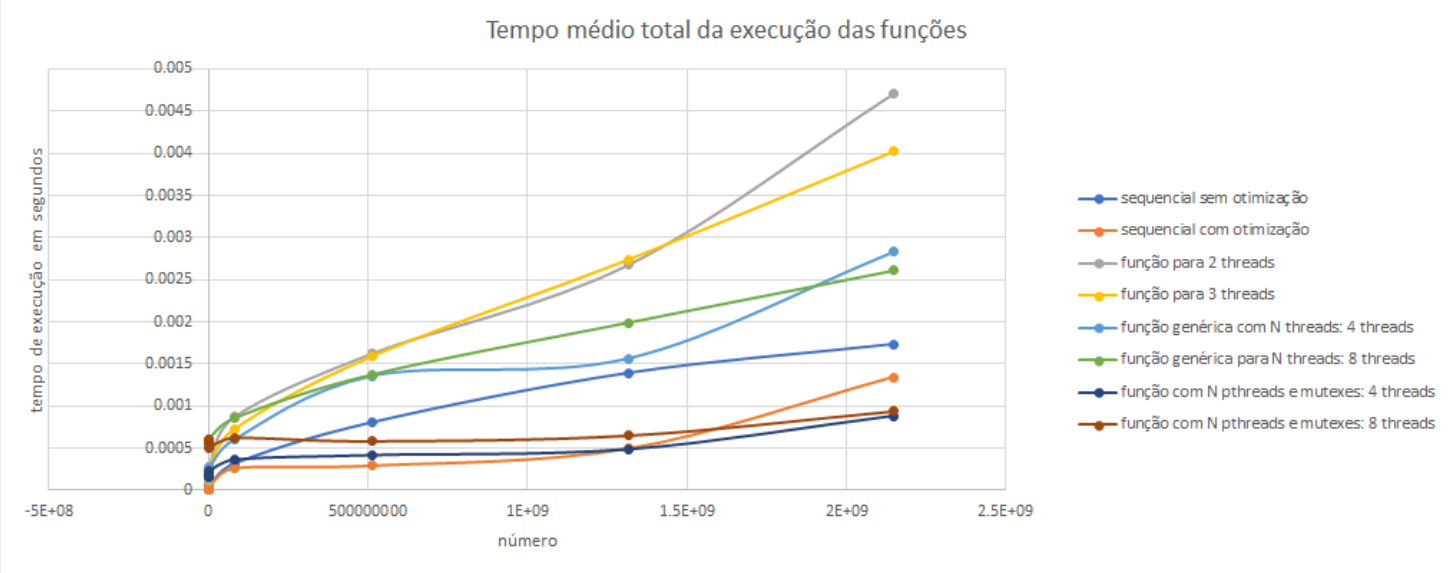


Gráfico 3 - tempo total de execução

Observando os dados obtidos no gráfico 3, podemos induzir que:

- Apesar das funções genéricas de paralelismo serem mais rápidas a executar a função dos factores, se estas não forem bem construidas, o seu tempo de execução total será elevado, como se pode ver, as funções com mutexes são muito mais rápidas a executar que as sem mutexes.
- Também se pode observar nas funções de processamento em paralelo, as funções que executam com 4 *threads* são ligeiramente mais rápidas que das de 8 *threads*, no entanto o

crescimento das funções de 8 *threads* é bastante inferior ao de 4 *threads*, por exemplo para a última implementação o crescimento de 4 *threads* vs o de 8 *threads* é de aproximadamente 334% a mais, o que poderá indicar que para números superiores ao testado, as 8 *threads* são de facto mais eficientes que 4

5.2.4. Aceleração das funções de threads em relação às sequênciais

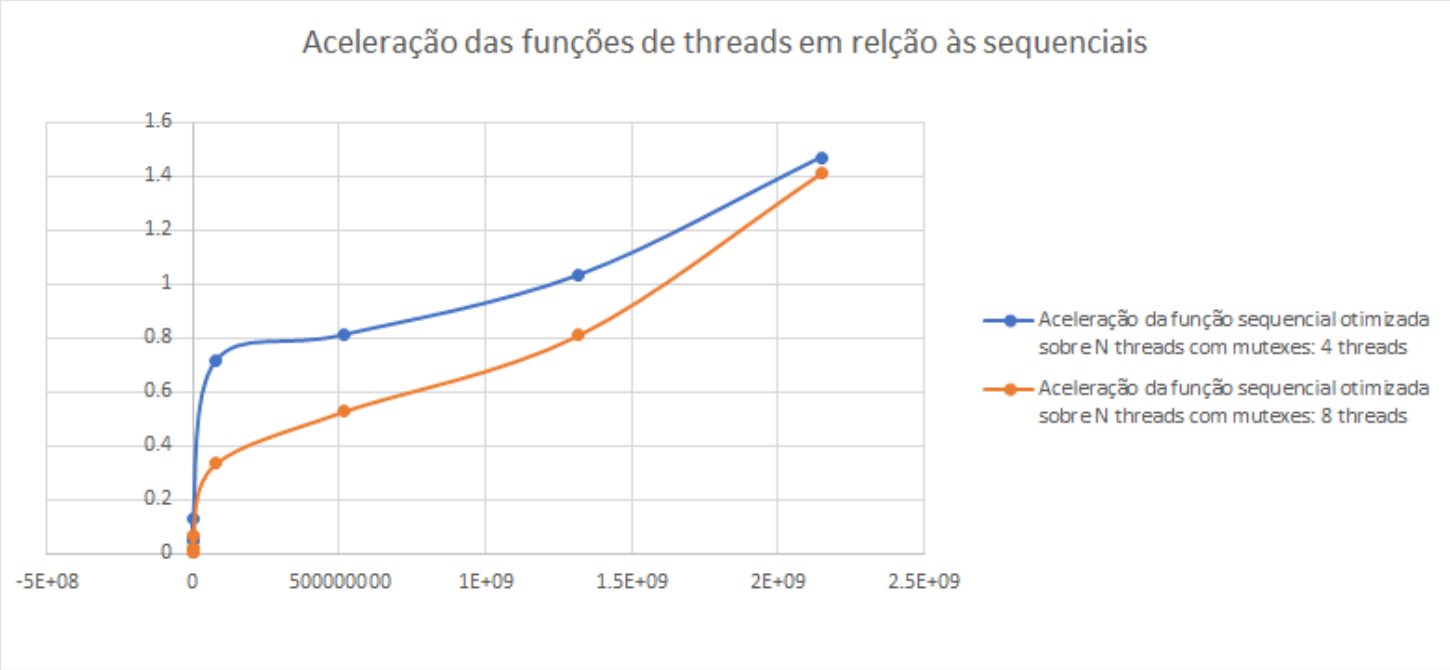


Gráfico 4 - Aceleração das funções de threads

Observando os dados obtidos no gráfico 4, podemos induzir que:

- As funções de *threads* só são mais rápidas que as funções sequenciais a partir de números na ordem dos 10^9, no entanto têm um crescimento linear a partir desse número

5.2.5. Eficiência das funções de threads em relação ao número de processadores

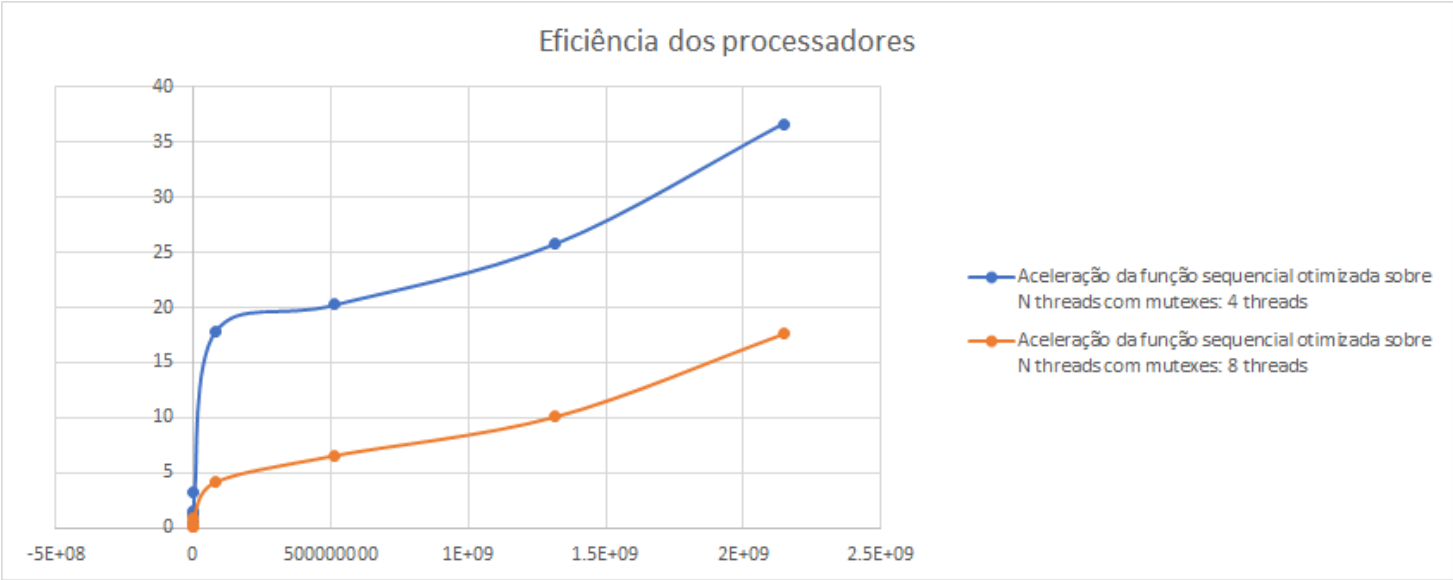


Gráfico 4 - Eficência das funções de threads

Observando os dados obtidos no gráfico 5 e da tabela 12, podemos induzir que:

- As função de 4 *threads* é 2 vezes mais eficiente que a de 8 *threads*

Observações finais:

- Comparando os valores das tablelas 9 ou 10, podemos observar que para as funções sequenciais a divergência entre o tempo gasto total e o tempo gasto pelo cpu são bastante semelhantes enquando nas funções são bastante inferiores, isso deve-se ao facto de system CPU time ser o tempo total de execução das funções dentro do núcleo, mas como estas funções são executadas em paralelo, então o user CPU time é apenas uma fração do tempo de execução total dentro do CPU. Podemos então ver que o CPU de facto desperdiça mais tempo a executar as funções em paralelo, no entanto estas estaõ divididas por *threads* o tempo total é bastante inferior, para o caso de N threads com mutexes: 4 threads podemos observar que a velocidade ganha é $\frac{0.203}{0.095} = 2.13$ o que significa que a função foi pelo menos 2x mais rápida do que se tivesse sido realiza em sequencial.

6. 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*, no nosso caso era só dividir o número total de operações pelas múltiplas *threads* e atualizar o array que guarda os resultados finais com mutexes. Podemos que concluir que o processamento paralelo apesar de ser mais complicado a implementar é muito mais rápido em termos de processamento para resolver problemas que requerem grande quantidade processamento computacional e que para cada problema também temos que ter em consideração o número das *threads* versus a complexidade do problema, o que seria um caso de estudo interessante.

7. Referências

[1] como utilizar "opções" nos comandos
[2] como utilizar multiplas opções associadas a um comando
[3] como criar um foreach em c
[4] como converter um char para int em c, ver reposta de Paul Tomblin
[5] como criar um array de funções, ver resposta de Swedgin
[6] função de tempo da Professora Margarida Moura, exericio 5
[7] implementação do Merge Sort em c
[8] função de clonar strings do Professor Pedro Guerreiro
[9] converter string para long, ver resposta de Spikatrix
[10] implementação do insertion sort em c
[11] exemplo de fazer um sumatório devalores de um array com pthreads
[12] exemplos de como criar e utilizar pthreads
[13] calculadora de factores
[14] videos de como calcular os fatores com 2, 3, 4, 5, 6, 9 e 10 da Khan Academy
[15] como escrever um relatório técnico
[16] imagem do cpu
[17] processamento em paralelo e threads
[18] API
[19] POSIX
[20] Gerar C-Docs
[21] Lista de como escrever funções em Markdown
[22] qsort
[23] Pdfs de Avaliação de Performance da Professora Margarida Moura da cadeira de SPD
[24] Definição de eficiência e aceleração da professora Margarida Madeira