Parallel Quicksort

Diogo Pereira, Luís Lopes

Universidade Do Minho, Portugal

Keywords: Quicksort, openMP, parallel, sequencial.

Abstract

The following report consists in a catching up of the work realised by students Diogo Pereira and Luís Lopes in order to improve and create a parallelized version of the quicksort algorithm using openMP and written in C, in order to optimize and speed up the sorting process using *multi-threading*.

1 Introdução

Produzir código eficiente é uma batalha constante para qualquer programador, existindo sempre espaço para melhorias seja no tempo de execução ou até na diminuição de recursos utilizados. Esta batalha pode ser combatida com recurso a *multi-threading* que consiste em paralelizar a execução do nosso código, dividir o mesmo por várias threads pode mostrar-se uma solução extremamente eficaz. Neste trabalho, realizado no âmbito da Unidade Curricular Computação Paralela, inserida no 4º ano do Mestrado Integrado de Engenharia Informática, é abordado o algoritmo quicksort, utilizado em várias bibliotecas das mais usadas linguagens de programação, **qsort** em C ou simplesmente **sort()** em Java. Neste relatório segue-se os passos seguidos para obter uma versão paralelizada deste algoritmo, seguindo como base uma implementação sequencial fornecida pelos professores, bem como uma análise de resultados deste processo.

2 Considerações Iniciais

2.1 O que é o algoritmo Quick-Sort

O algoritmo quicksort foi desenvolvido em 1960, por Tony Hoare. Tony Hoare é um professor emérito da Universidade de Oxford e foi na busca de uma melhor performance no *lookUp* a um dicionário de palavras que descobriu o algoritmo quicksort [1] . Este algoritmo resume-se na teoria de que um problema é mais rápido de resolver se o podermos dividir em dois sub-problemas, e esses sub-problemas serem reduzidos a problemas ainda mas simples. O quicksort é então resolvido seguindo dois passos **partição** e **recursividade**. Começa-se então pela partição, idealmente escolheria-se um valor, que será o pivot, que estará numa posição em que todos os elementos anteriores a este são menores e todos os elementos do vetor à direita deste serão maiores. No entanto a existência de um pivot que separe desta forma o vetor é rara e mesmo que acontecesse, a sua posição não seria definida. Porém é possível reordenar o array de forma a que isso aconteça. Escolhido então o pivot, de forma arbitrária, começa-se por percorrer o vetor desde a sua primeira posicação até encontrar um valor superior**(lo)** ao pivot, de seguida vamos começar pela ultima posição do vetor e percorrer, no sentido inverso, até encontrar um valor inferior**(hi)** ao pivot. Decorrida esta primeira fase vamos fazer *swap* aos valores que se encontram nas posições **lo** e **hi**, se **hi**$<$**=lo** e repete-se o ciclo até à condição anterior não corresponder à verdade. Seguindo estes passos obtém-se então um vetor organizado e divido em valores $<$**pivot** e $>$**pivot**. Então vamos aplicar recursividade aos dois vetores mais pequenos, formados através da divisão. Assim acabamos com um vetor ordenado de forma rápida e efieciente [2] .

2.2 Caracterização das Máquinas Utilizadas

De forma a obter uma amostragem razoável e heterogénea utilizamos várias máquinas com CPUs diferentes. As quais podemos ver e analisar na tabela 1.

Máquina	CPU	GPU	RAM
GL551	i7-6700HQ	GTX960M	16GB
L-Fixo	Core2Quad Q9550	GT710M	6GB
MSI	i7-10710U	GTX 1650	16GB
D-Fixo	i5-7500HQ	GTX1080	16GB
Macbook	Apple M1	Apple M1	16GB
Cluster	———	———	—-

Table 1. Máquinas utilizadas.

3 Diferentes Implementações

3.1 Implementação Sequencial

```
void quicksort(int *array, int lo, int hi){
int i=lo,j=hi,h;
    int x=array[(lo+hi)/2];
    //partition
    do  {
        while(array[i]<x) i++;
        while(array[j]>x) j--;
        if(i<=j){
        h=array[i];
        array[i]=array[j];
        array[j]=h;
        i++; j--;
```

```
        }
    }while(i<=j);
    //recursion
    if(lo<j) quicksort(array,lo,j);
    if(i<hi) quicksort(array,i,hi);
}
```

De forma a seguir o referido em 2.1, conseguimos dividir o código em duas partes. Primeiro começamos por escolher o pivot que terá o seu valor armazenado em x e fazemos a partição com recurso a um ciclo **do while**, após isso é invocada a recursividade. Esta implementação, apesar de rápida torna-se ineficiente quando o tamanho do array toma porpoções maiores. Nos resultados obtidos podemos verificar uma diminuição de tempo de execução de até 5x.

3.2 Implementação Paralela

```
void quicksort(int* array,int lo,int hi){
int pivotElement;
if((hi - lo + 1) < 10000) {
serial_quicksort(array,lo,hi);
}
    else {
int i=lo,j=hi,h;
    int x=array[(lo+hi)/2];
do{
        while(array[i]<x) i++;
         while(array[j]>x) j--;
         if(i<=j){
         h=array[i];
         array[i]=array[j];
         array[j]=h;
         i++; j--;
         }
     }while(i<=j);
#pragma omp task
{
quicksort(array,lo,j);
}
#pragma omp task
{
quicksort(array,i,hi);
}
}
}

int main(int argc, char* argv[]){
    //...//
    #pragma omp parallel
#pragma omp single
{
    quicksort(arr, 0, arraySize-1);
}
    //...//
}
```

Como referido na explicação do algoritmo quicksort, este é por duas partes e foi necessário, antes de fazer qualquer alteração, perceber onde o podiamos otimizar. Ora, é necessário primeiro fazer o rearranjo ao vetor e esta operação não benefeciaria de ser executada em paralelo. Desta forma será mais lógico atribuir as chamadas recursivas como tarefas a serem executadas. Até chegarmos à implementação mais eficiente e correta foi necessário passar por várias versões. Contudo é necessário, também, perceber quando deixará de fazer sentido paralelizar, pois associado à atribuição de *threads* existe um custo que poderá em certos casos superar o tempo de execução da implementação sequencial, não se justificando assim a paralelização. Para tal após testes definimos que vetores com tamanhos inferiores a 10000 inteiros não seriam vetorizados.

3.2.1 Revisão 1 - OMP Parallel Only

Numa primeira Revisão, além de definir as chamadas recursivas como tarefas com recurso ao *#pragma omp task*, a única alteração à implementação sequencial foi acrescentar o *#pragma omp parallel* antes da chamada ao algoritmo. Este comando ”explicitly instructs the compiler to parallelize the chosen block of code.”[3]. No entanto os tempos de execução encontrados seguiram o sentido aposto do desejado, visto que quase dobraram o valor da implementação sequencial, como podemos verificar no gráfico 1.

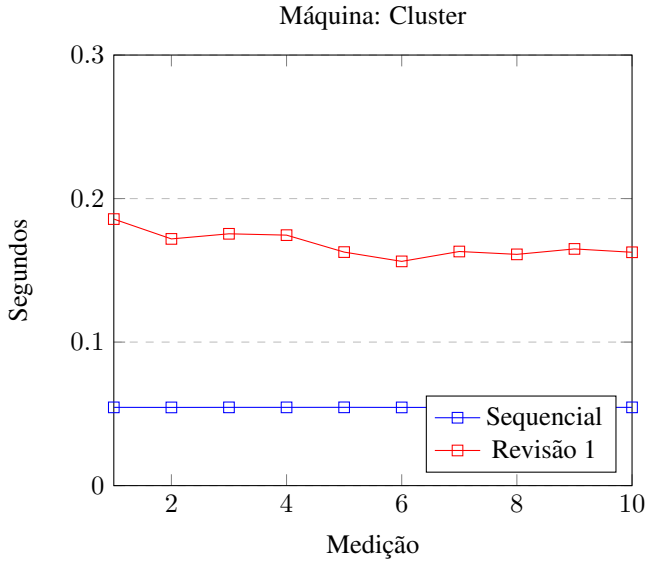

Chart 1. Comparação de tempo de execução sequencial, Revisão 1, utilizando o Array 1.

3.2.2 Revisão 2 - OMP Parallel and OMP Single

Como os resultados obtidos na primeira Revisão não foram de encontro ao esperado, é necessário continuar a implementar difentes diretrizes para melhorar os resultados. Desta forma decidimos utilizar o *#pragma omp single*, que alterou significamente o cenário anterior e foi de encontro a medições bastante melhores, revelando que este seria o percurso correto a seguir, como podemos verificar no gráfico 2

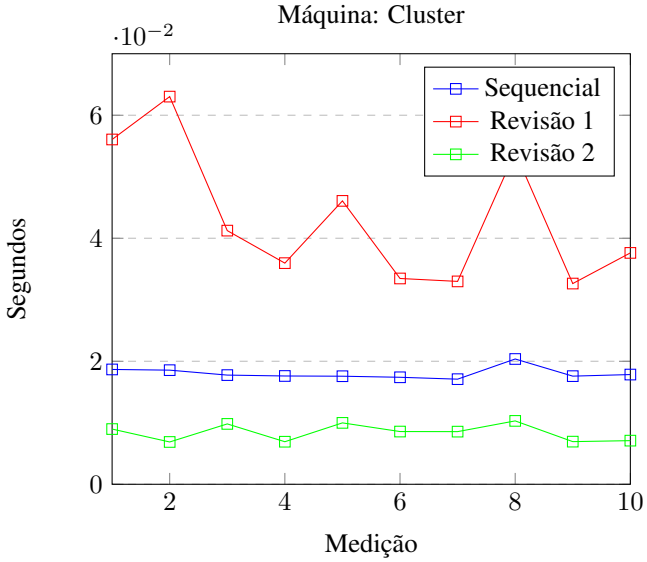

Chart 2. Comparação de tempo de execução sequencial, Revisão 1 e Revisão 2, utilizando o Array 1.

3.2.3 Revisão 3 - OMP Parallel, OMP Single and OMP Taskwait

Apesar de já conseguirmos obter tempos consideravelmentes melhores, achamos que existia espaço para melhoramentos e estabilidade. Nesta ótica decidimos introduzir o *#pragma omp taskwait*. Esta diretiva "specify a wait for child tasks to be completed that are generated by the current task."[3]. Apesar de o objetivo ser minimizar o tempo de execução, este não foi cumprido, pois esta ação levou a um acrescimento de tempo de execução em todas as máquinas testadas.

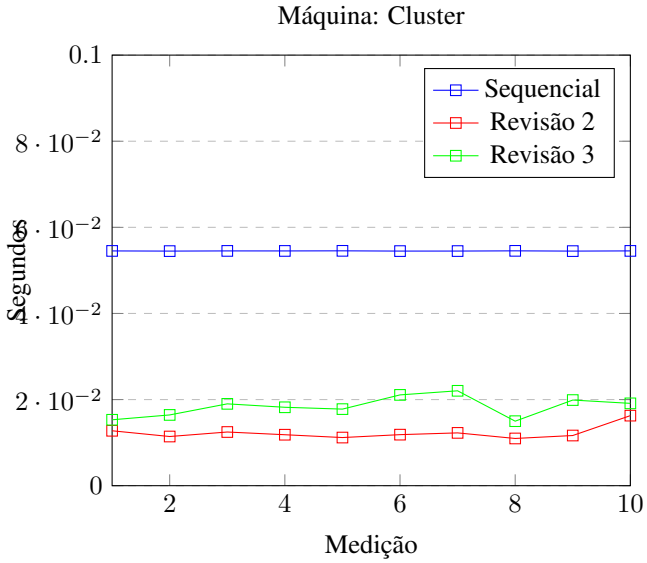

Chart 3. Comparação de tempo de execução sequencial, Revisão 2 e Revisão 3, utilizando o Array 1.

3.2.4 Revisão 4 - OMP Parallel, OMP Single, and Default(Shared)

Como na revisão 3 o tempo de execução aumentou, decidimos voltar atrás e continuar com a implementação da revisão 2. Na procura de otimizar então, novamente, a revisão 2 foi implementado o acrescimo de parâmetros opcionais ao *#pragma omp task*. A adição do parâmetro *default(shared)* que consiste em definir o *data scope* padrão da variável em cada task[3], ao referir que serão *shared* estamos a afirmar que cada variável estará numa *shared(list) clause*[3]. Contudo esta revisão encontrou diferentes resultados em diferentes máquinas. No Cluster, por exemplo, a adição deste parâmetro levou ao acrescimo do tempo de execução. Por outro lado no MSI ou no D-Fixo conseguimos verificar ligeiras melhorias.

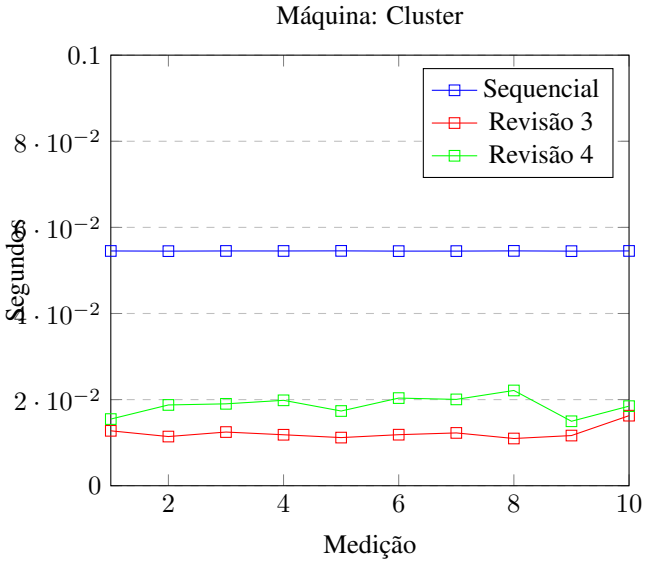

Chart 4. Comparação de tempo de execução sequencial, Revisão 3 e Revisão 4, utilizando o Array 1.

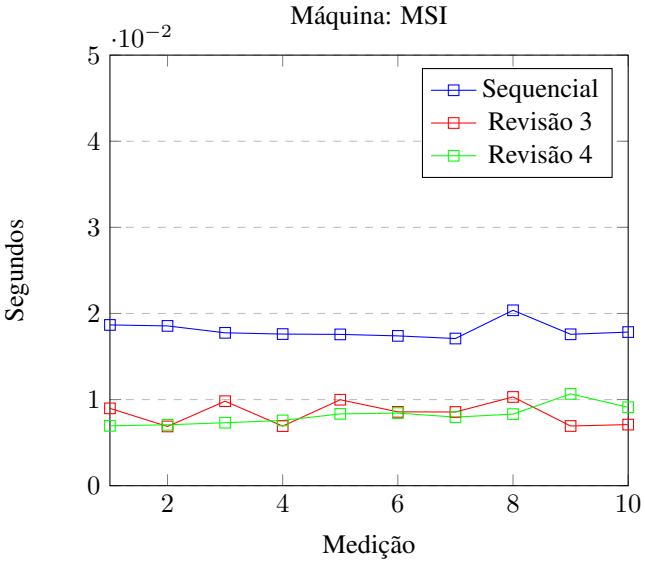

Chart 5. Comparação de tempo de execução sequencial, Revisão 3 e Revisão 4, utilizando o Array 1.

3.3 Ultimas considerações

De facto após várias revisões, podemos verificar que os métricas dependem, não só, da otimização implementada pelo programador, mas também pela máquinas e arquiteturas que as equipam. Porém, pela sua potencialidade e credibilidade, os valores aos quais depositamos mais confiança e utilizamos para uma decisão final serão os fornecidos pelo Cluster, deste modo a implementação final, reflete-se na revisão 2, que apresentou os melhores resultados nos testes efetuados no Cluster.

4 Conjunto de Dados

O tempo de execução do quicksort depende, além da implementação e da máquina em que é executado, também, do *input* que recebe. Inicialmente foram utilizados vetores que eram alocados e preenchidos aleatoriamente antes de executar a sua ordenação. Ora isto constituia um problema para a válidação das métricas obtidas, isto porque se os vetores são diferentes o tempo de execução será, à partida, diferente. Para que este não fosse um problema que afetaria as métricas e medições efetuadas, criamos quatro vetores que foram gerados, de forma aleatória e, guardados num ficheiro.txt para que em todas as comparação de tempo de execução insidissem sobre o mesmo *input*. Neste sentido antes da execução do algoritmo foi implementada uma função *void readFile(FILE *fp,int *array, int arraySize)*. É importantes salientar que todas as chamadas a funções ou atribuição de variáveis que ocorram antes ou após o quicksort, não contam para a conometragem do tempo. Assim para as medições foram utilizados quatro vetores de diferentes dimensões para que se possa comparar os ganhos de eficiência consoante o tamanho do vetor. Temos então o Array 1 com **quinhentos mil** valores, Array 2 com **duzentos e cinquenta mil** valores, Array 3 com **cem mil** valores e por fim Array 4 com **cinquenta mil** valores.

5 Análise de Resultados

Quanto maior for o vetor, maior será o seu tempo de ordenação, assumindo que não estava previamente ordenado. Ora com a paralelização o mesmo se verifica, sendo que o Array 1 é o que leva mais tempo a ordernar em todas as revisões e mesmo na versão sequencial. Além de ser o que, por razões obvias, maior tempo de excucação, é, também, onde se reflete maior a potencialidade e os ganhos associados à paralelização de código. Podemos verificar pelo gráfico 6 que os ganhos, apesar de noutros vetores chegarem a 50%, no Array 1 ultrapassa esses valores, tendo em média reduzido em mais de metade o seu tempo de execução. O gráfico apresentado indica os tempos médios lidos na implementação sequencial e na revisão 2, com a atribuição de 16 threads, disponíveis no cluster. Além do gráfico existe uma folha de cálculo, não editável, anexada ao relatório que contem todas as medições e métricas obtidas.

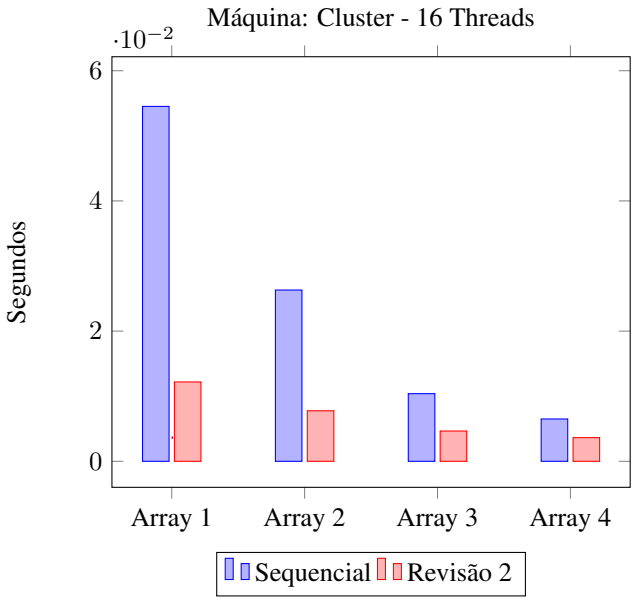

Chart 6. Comparação dos ganhos entre os diferentes arrays.

5.1 Ambiente Multi-Thread, quantas threads usar?

Aproveitando um bocado de toda a potência que o Cluster nos tem para oferecer, foram efetuados testes para verificar os ganhos ou perdas com a atribuição de diferentes números de *threads*. Este número tomou então os valores 2, 4, 8, 16, 32 e 48 *threads*. Com a análise dos valores apresentados no gráfico 7, conseguimos verificar que consoante o aumento de *threads*, até as 16, o tempo de execução diminui, no entanto após as 16 *threads* o tempo de execução aumentou, assim podemos afirmar que será melhor correr o algoritmo com 16 *threads*. Através do gráfico 7 conseguimos ter uma melhor percepção das diferenças causadas.

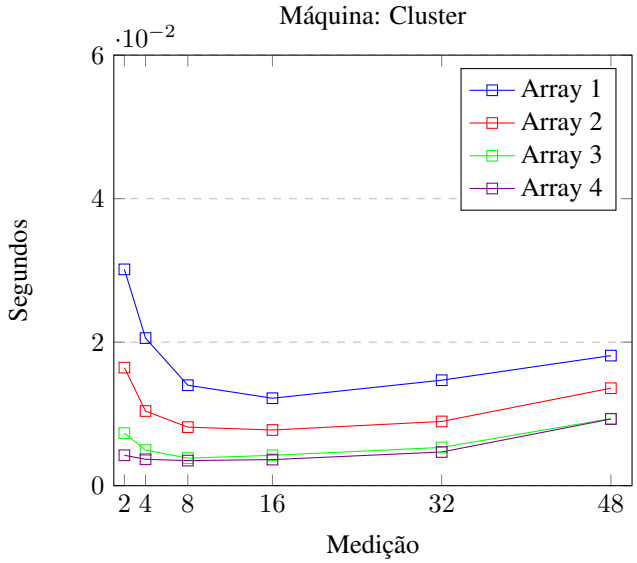

Chart 7. Comparação de tempo de execução sequencial, Revisão 3 e Revisão 4, utilizando o Array 1.

5.2 Validação dos dados

A utilização de diferentes máquinas permite-nos ter uma maior confiança na implementação final, devido aos extensos dados recolhidos e permitiu fazer um pequeno estudo sobre o impacto da paralelização, as diferenças entre as arquiteturas, bem como o desenvolvimento que ocorreu nos ultimmos anos nos processadores. Na amostragem utilizada podemos encontrar processadores lançados no mercado no primeiro trimestre de 2008 (Core2Quad), como processadores lançados no último trimestre de 2020 (Apple M1).

5.3 Comparação Máquinas

Como seria de esperar o Core2Quad Q9550, com os seus 4 cores, transistores de 45nm, 12 MB L2 Cache e 2.83 GHz de clock foi o que obteve os piores tempos de execução, apesar disso, surpreendenemente, foi também o que tirou mais proveito da paralelização, como referido no ponto 3.1 o tempo de execução passou para cerca de 25% com a implementação paralela. Outra previsão que aconteceu foi a superioridade dos processadores de acordo com a sua geração. Tudo isto é observável no gráfico 8 e nos ficheiros anexados.

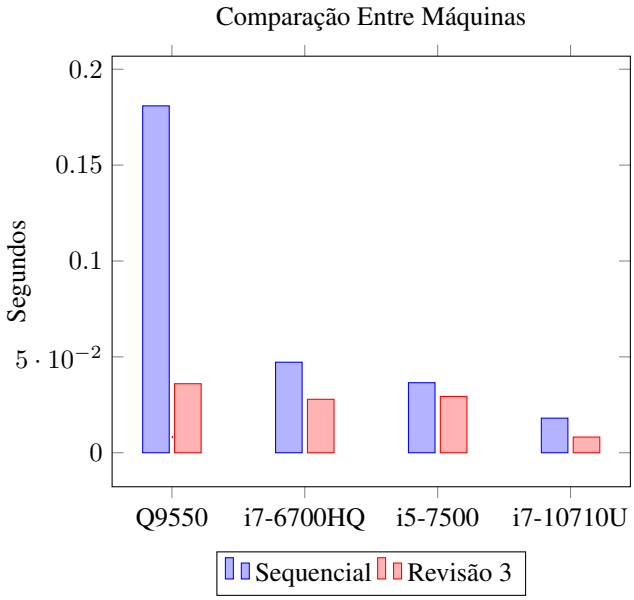

Chart 8. Comparação dos ganhos entre as diferentes máquinas, com arquitetura Intel, usando o Array 1.

6 Intel vs Apple

Durante as medições e o desenvolvimento deste trabalho, surgiu a oportunidade fazer testes, tanto em processadores mais antigos, como processadores bastante recentes. Desta forma, orgulhosamente, conseguimos efetuar testes com o mais recente e primeiro processador desenvolvido em arquitetura ARM por uma das maiores, senão a maior, empresa tecnológica, Apple. Neste seguimento, conseguimos comparar os valores obtidos numa tecnologia Intel *vs*, tecnologia Apple. Verificamos então, na tabela 2, que o Apple M1 possui efetivamente mais cores e transistores de bastante menor tamanho, cerca de quase 1/3.

	Apple M1	Intel i7-10710u
Lithography	5nm	14 nm
Transistors	16000M	———
Cores	8	6
Cache	12MB	12mb
Max memory bandwidth	68.25 GB/s	45.8 GB/s
Base Clock	2.1GHz	1.10 GHz

Table 2. Máquinas utilizadas.

O que significa que no mesmo espaço que a Intel, a Apple consegue colocar cerca de 3x mais transistores, o que leva, automaticamente a um melhor nível de desempenho. Todas as caracteristicas mencionadas levariam a afirmar que, trivialmente, o Apple M1 superaria os valores obtidos pelos processador i7-10710, porém tal não acontece. Acreditamos que tal resultados, visíveis no gráfico 9, devem-se ao facto de todo o sistema operativo, bem como compiladores e todas outras variantes ainda não estarem totalmente otimizadas para receberem este novo *system-on-a-chip*(SoC).

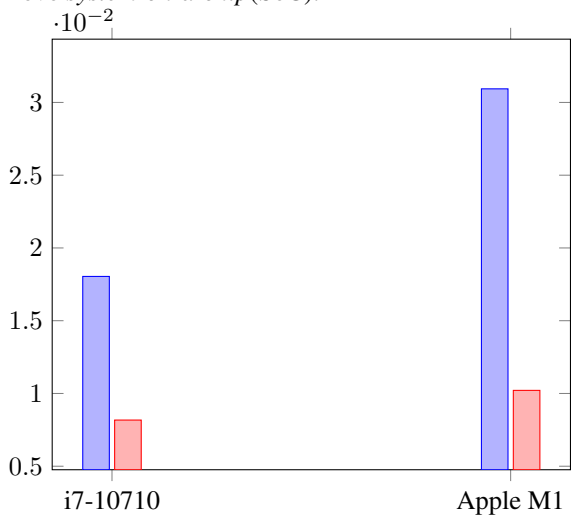

Chart 9. Comparação dos ganhos entre Intel vs Apple, usando o Array 1.

7 Conclusão

Ao longo deste documento explicaram-se os passos que foram seguidos para obter uma implementação paralela do algoritmo quicksort. Considera-se que os tempos de execução obtidos foram bons e dentro do esperado. Afirma-se também que a utilização de diferentes máquinas, baseadas em diferentes arquiteturas, com também propósitos diferentes, foram um ponto fucral para a validação e melhor percepção dos resultados obtidos. Deste modo foi perceptível que existem várias alternativas e que cada máquina é diferente reagindo de forma diferente. Contudo é possível também tecer críticas ao número de testes efetuados, apesar de terem sido utilizados 4 vetores de diferentes tamanhos e serem medidas sempre 10 execuções por cada implementação, nada prejudicaria efetuar mais medições com mais *inputs*. Dito isto e considerando o panorama geral pode-se afirmar que este trabalho foi completado com sucesso.

References

[1] D. of Computer Science, "Tony Hoare." https://www.cs.ox.ac.uk/people/tony.hoare/. [Online; accessed 23-Jan-2021].

[2] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, p. 10–16, 1962.

[3] IBM, "Pragma directives for parallel processing." [Online; accessed 23-Jan-2021].