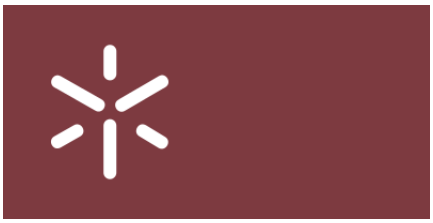

Universidade do Minho

Mestrado em Engenharia Informática

Engenharia de Sistemas da Computação

2018/2019 - Trabalho Prático 2

Paralelização do algoritmo Heap Sort

Autores:

André Pereira	PG38923
Vasco Leitão	PG38429

10 de Junho de 2019

Conteúdo

1 Introdução

Neste relatório, no âmbito da UC de Engenharia de Sistemas da Computação, será abordada uma solução paralela para o algoritmo de ordenação *Heap Sort*, usando programação *multithreading*.

O objetivo deste projeto é a implementação de um algoritmo que ordena uma lista de elementos através do método *Heap Sort*, através da incialização de múltiplos fios de execução. Dados este paradigma e a natureza deste algoritmo de *sorting*, terão também de ser explorados diferentes mecanismos de sincronização, fazendo uso de semáforos/*locks* e tipos de dados especiais construídos para o efeito.

Assim, foram desenvolvidas duas versões deste algoritmo, recorrendo a linguagens de programação diferentes, explorando assim este problema com estratégias distintas. Cada uma destas utiliza mecanismos diferentes na abordagem das zonas de exclusão, mantendo assim a aplicação do algoritmo de *sorting* livre de *data races*.

Por fim, serão apresentados diferentes medições e métricas, que incluem os quadros de resumo dos tempos de execução e escalabilidade. Os resultado obtidos serão descritos e explicados também com recurso a uma ferramenta de medição - o `perf`.

2 Heap Sort

2.1 Algoritmo

O *Heap Sort* é um algoritmo de ordenação de listas. Este algoritmo suporta-se da estrutura de dados *heap*. Uma *heap* é uma árvore binária em que os valores dos filhos de um nodo são sempre mais pequenos do que o próprio, assim a raíz desta árvore contém o maior elemento da mesma. Assim para se fazer fazer a ordenação basta retirar a raíz desta desta *heap* sucessivamente até ficar vazia adicionando este elemento na ultima anterior á posição do último elemento retirado, no caso de se tratar de uma *max heap*. De salientar que quando se retira raíz da *heap* é necessário ocupar esse lugar, para isso, faz-se uma troca com o último elemento da árvore, mas assim a *heap* não preserva a propriedade de o elemento na raíz ser o maior elemento da árvore, é então necessário colocar este elemento no sítio coreto, isto é, trocando-o com os filhos (*bubble-down* até ao caso em que ambos são mais pequenos.

Assim temos que o tempo de execução deste algoritmo é dado pelo tempo de transformar a lista numa *heap* mais n vezes, o necessário para colocar o elemento da raíz na posição correta, vezes o tempo desta operação. O tempo de transformar esta a lista numa *heap* é assimptóticamente: $\mathcal{O}(n)$, colocar a raíz no sítio correto é no pior caso percorrer apenas um caminho desde a raíz até às folhas e uma vez que se trata de uma árvore binária o comprimento deste caminho é dado pela expressão: $\log_2 n$. Portanto o tempo de execução total deste algoritmo é dado pela expressão: $\mathcal{O}(n) + n \times \mathcal{O}(log_2 n)$, ou seja, um tempo assimptótico: $\mathcal{O}(n.\log_2(n))$

De notar que esta árvore tem a particularidade de poder ser armazenada num *array* em que os filhos da posição *parent* são obtidos através das seguintes expressões $left = i \times 2 + 1$ e $right = i \times 2 + 2$.

2.2 Paralelização

Heapsort - versão C++

Foi desenvolvida uma primeira versão para a resolução do problema, utilizando `C++14` que dispõe de bibliotecas *multithreading* e de gestão de memória, sendo que existem diferentes fios de execução a efetuar operações sobre os mesmo recursos.

O algoritmo começa por traduzir uma lista de inteiros gerados aleatoriamente numa *heap*, que é armazenada numa estrutura de dados do tipo *array*. Este array é de seguida ordenado sequencialmente, segundo um critério definido - contruir uma *max Heap*, de modo a que se obtenha uma árvore binária, onde todos os elementos abaixo de um nodo são inferiores a ele próprio. Esta propriedade é importante, pois ao longo do processo queremos manter o *array* desta forma, garantindo assim que o primeiro elemento do (a raíz da *heap* é o maior elemento existente.

A paralelização começa de seguida, onde são criadas N threads que vão acedendo de forma sequencial [1] a *heap*. Cada *thread* está encarregue apenas

[1] Diga-se sequencial no sentido que uma *thread* nunca ultrapassa outra à medida que vai

de duas tarefas: trocar o primeiro elemento da árvore que é garantidamente o maior quando o acede, com o da última posição que não foi trocado do *array* (inserindo assim os maiores elementos no *array* da direita para a esquerda, de modo a formar uma lista com ordem crescente); a seguinte tarefa é colocar no sítio correto este novo elemento que foi substituído e ficou na raíz da *heap*, assim o objetivo é garantir que a subárvore que a *thread* trabalhe seja uma *maxheap*.

Dado estes múltiplos acessos, temos um problema de concorrência sobre o mesmo objeto partilhado por todos os fios de execução - a `heap`. Para resolver este problema, cada *thread* ao aceder pela primeira vez à *heap* (à raíz desta), utiliza um *mutex*, tentado assim adquirir o *lock* do indice 0 do array. Existe um *array* de N mutexes, cada um destes referenciados para o acesso de cada um do nó da árvore. Assim cada fio de execução inicialmente tenta adquirir o *lock* da raíz da árvore e do indice onde este elemento será introduzido no array final. É utilizada uma varíavel atómica para guardar o índice a ser introduzido este atual maior elemento da *heap*, de forma a garantir atomicidade nas operação de subtração, quando os elementos são trocados, e então decrementada esta varíavel, prevenindo assim que diferentes *threads* decrementem em simultâneo. Ora, existindo extão o *lock* destes dois elementos, estes são trocados, inserindo então este atual maior elemento da *heap* no *array*. Após isto a *thread* vai empurrando este novo elemento que ficou na raíz para baixo, caso este elemento seja inferior que cada um dos filhos, para isto nesta segunda fase, a thread que tem lock realizado neste nó, tentará fazer *lock* dos filhos, e quando adquirido, comparará os seus valores, trocado o nó com o maior filho; se não for necessário trocar, esta *thread* é terminada, e a meio deste processo, poderiam ter entrado novos fios nos nós acima destes, desde que consigam adquirir os seus *locks*. Assim, a *thread* vai fazendo *bubble down* do elemento, adquirindo sempre os *lock* dos novos filhos, e libertando o *lock* do nó anterior (o que era o maior filho). Assim à medida que uma *thread* vai efetuando o *bubble down* concorrente de um elemento, outra novas *threads* váo acedendo à *heap* sempre pela raíz, e efetuando estes mesmos processos, nunca ultrapassando uma outra *thread* no seu *bubble down*, garatindo assim a ordem das *threads* ao fazer a travessia pela árvore. No entanto uma *thread* pode ultrapassar o nível de uma outra desde que consiga adquirir os *locks*, dado que uma *thread* ao descer de nível nunca liberta os *locks* do nó atual e do seguinte em simultanteo, garatindo sempre um *lock* ativo; este fenómenos de ultrapssagem só é possível se for efetuado numa subárvore disinta, não afetando assim as propriedades e leis que foram definidas.

descendo pela árvore, fazendo *bubble down* do menor elemento.

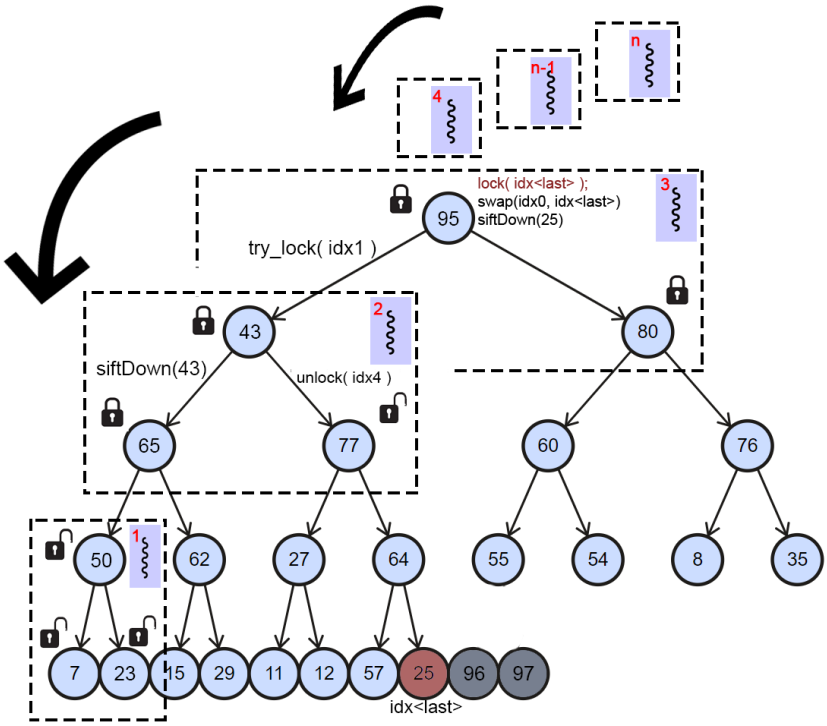

Figura 1: Comportamento das *threads* na *heap*

A imagem anterior, exemplifica este processo referido. Os elementos 97 e 96 já foram colocados no final da *heap/array* pela *thread* 1 e 2, respetivamente. A *thread* 1 fez *swap* na sua raíz do 97 com o elemento 23, tendo realizado o *bubble-down* deste elemento até atingir a últma subárvore de 3 elementos do lado esquerdo; é libertado o *lock* da posição onde se encontra o número 50 e o 7, e como o 23 manter-se-á nesse local também é libertada a posição. A meio deste processo foram entrando novas *threads* realizando este mesmo processo. A *thread* 2, substituiu o elemento 96, pelo 43, e atualmente tem o *lock* do seu nó e do 65 adquirido, pois está prestes a efetuar a sua troca. Entretanto também já entrou uma nova *thread* 3, que efetuou o *lock* do *mutex* da posição do 95 e do último elemento "disponível"da *heap* - o 25, estando prestes a fazer a primeira troca. Quando a *thread* 2 libertar o *lock* do 43, a *thread* 1 poderá adquiri-lo, de forma a comparar os filhos, e eventualmente trocar o 25 com o 80.

Heapsort utilizando tasks (JAVA)

Uma outra versão foi implementada recorrendo à utilização de *tasks* e padrão *master-slave*. Neste caso, uma vez que as versões mais antigas de `C++` não suportam a submissão de *tasks* para uma *thread pool* a implementação foi feita utilizando a linguagem de programação **Java**. Nesta implementação à semelhança da anterior é feita a transformação do *array* numa *heap* sequencialmente, e fazendo em paralelo as *tasks* que correspondem ao *siftDown* da raíz

depois de retirado o maior elemento, para que o *array* preserve a propriedade de *heap*.

Assim temos que, após a transformação do *array* numa *max heap* é criada uma *pool* de *threads*, com o número de *threads* desejadas, instanciado um objeto da classe `ExecutorService`, de seguida a *master* fica responsável por fazer a troca da raíz da *heap* para a posição do *array* indicada pela variável *end*, é declarada como sendo `volatile` pois também será utilizada pela *slave threads* que as *tasks* e portanto não era apropriado que estas guardassem esta variável em *cache*. Uma vez que a *thread master* apenas pode trocar a raíz após uma *task* garantir que na raíz está o maior elemento, a utilização de um simples *lock* sobre a raíz não era suficiente, pois a *thread master* poderia fazer *unlock*, criar a *task* e tornar a fazer o *lock* da raíz, sem a *task* ter começado. Assim foi necessária a utilização de uma predicado e de uma `Condition` sobre o `lock` da raíz par que a *thread master* possa fazer *await* e as *slave threads* possam sinalizar a verificação da propriedade da *heap* no nível da raíz e assim a *thread master* poder criar uma nova *task*, de notar que o teste desta condição foi feito através de uma *while* e não de um *if* devido á possibilidade de ocorrer um *spurious wakeup*.

Para além disso foi também feita atenção à possibilidade das *threads* que executam as *tasks* poderem ultrapassar outras começadas primeiro, no caso de estas seguirem por sub-árvores diferentes. Isto é, no final das iterações do *loop* executado no *siftDown* de um elemento é feito apenas o *unlock* do pai e do filho que não foi utilizado no *swap*, garantindo assim que outras *threads* não possam fazer o *lock* do elemento que se fez *swap*, na iteração seguinte como normalmente, bloqueado de novo os filhos desse elemento, e tornando-se assim este elemento o pai. Assim na saída desta rotina é apenas necessário fazer o *unlock* do próprio. De notar por fim que sempre que se realiza uma troca de elementos por parte das *tasks* é necessário verificar que a posição da troca pertence à *heap* e não ao *array* ordenado, uma vez que a cada task criada a *heap* diminui. Assim é necessário fazer o *lock* da variável *end* antes de se realizar qualquer *swap*, ou qualquer escrita nesta variável e de seguida o respetivo *unlock*, para isso utilizou-se o método *synchronized* sobre a variável *end* com estas operações dentro do *scope* criado.

3 Análise dos resultados

3.1 Heap Sort - Java

As medições realizadas, deste algoritmo de ordenação, forma feitas utilizando uma lista de com um milhão de elementos do tipo `int`, de seguida serão apresentados três gráficos sobre a escalabilidade do algoritmo, e sobre as *cache-misses*, fazendo variar o número total de *threads*.

O gráfico da escalabilidade apresentado é da escalabilidade forte, isto é, utilizando os mesmos dados, fazendo variar apenas o número de *threads*. Como era de esperar o algoritmo não escala, ou seja, o *speedup* deste algoritmo não ultrapassa o valor um. Pelo contrário, também não existe uma diminuição da $performance$, derivada do maior *overhead* de uma maior paralelização, o que se deve a este algoritmo ser executado com recurso à **JVM** e esta fazer otimizações

e a gestão adequada das *threads*.

As razões que se identificaram para a não escalabilidade do algoritmo foram, que apesar da árvore que representa a *heap* poder ser representada num *array*, os acessos a esta são irregulares, o que leva a um elevado número de *cache-misses*, neste caso o uso de várias *threads* também não faz um melhor uso das *caches*. Pelo contrário, uma vez que se utilizam *lock's* de forma a evitar *data-races* dos vários elementos do *array*, é necessário garantir a consistência da memória e portando é necessário invalidar as linhas de *cache* das outras *threads* onde se encontram esses elementos. Aumentando assim as *cache-misses* com o número de *threads*.

Assim sendo foi utilizada a ferramenta **perf** para realizar as medições das *cache-misses*, e como era de esperar a percentagem destas aumentou com o número de *threads*, tendo esta uma percentagem de aproximadamente 4% na versão sequencial e uma percentagem de aproximadamente 24% quando executado com dezasseis *threads*, sendo este um aumento substancial o algoritmo não poderia escalar pois os custos de acesso à memoria são muito elevados.

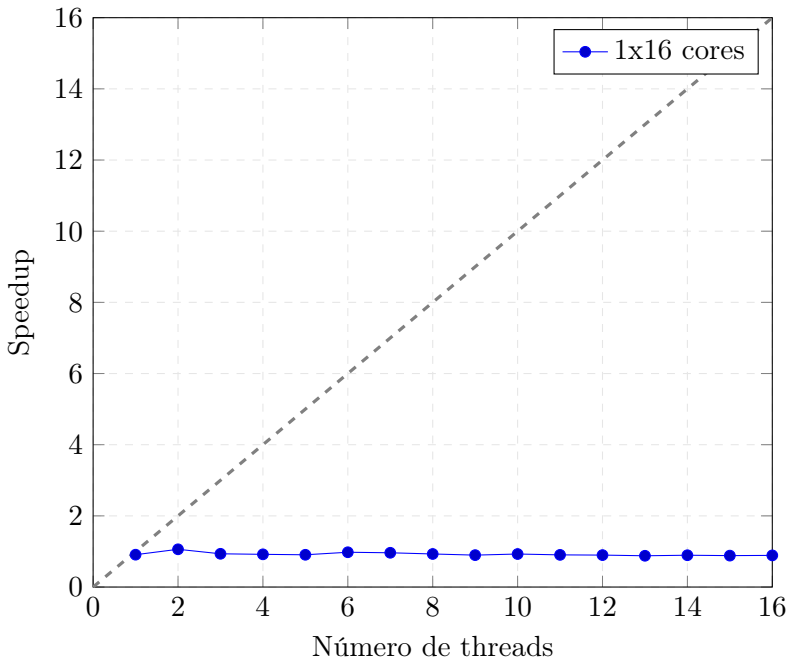

Figura 2: *Speedup* heapsort desbloqueando sub-árvore não utilizada

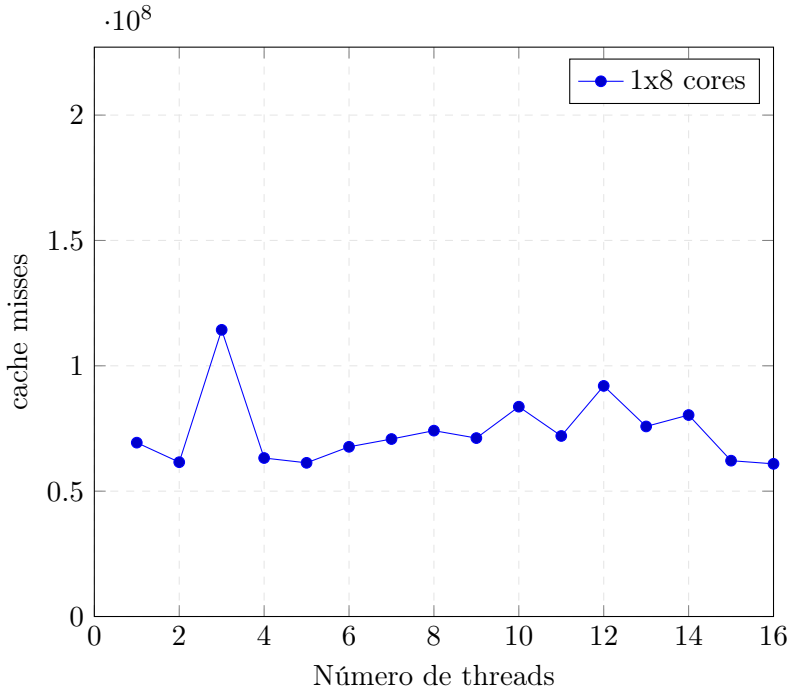

Figura 3: Número de *cache-misses* da heapsort em Java

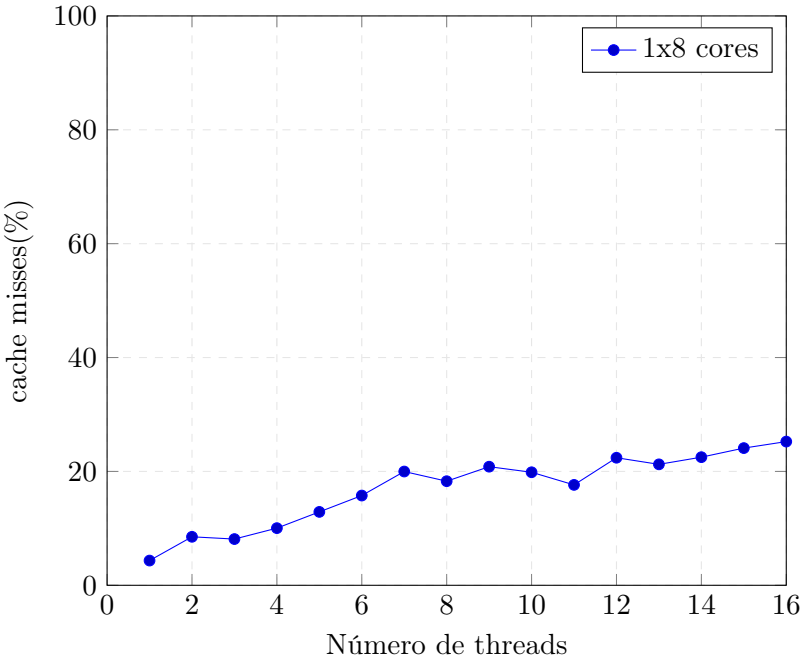

Figura 4: Percentagem de *cache-misses* da heapsort em Java

3.2 Heap Sort - C++

De seguida são apresentadas as medições para a versão desenvolvida exclusivamente por *threads*. Estas medições não foram realizadas no *cluster SeARCH* pela indisponibilidade do mesmo. Tendo sido testado num computador pessoal que possui quatro cores físicos, com especificações mais detalhadas em anexo.

O perfil de execução é apresentado de seguida, sendo composto por três gráficos, cujos demonstram o *speedup* da versão paralela e o número e percentagem de *cache-misses*.

Esta versão à semelhança da apresentada anteriormente também não tem um desempenho favorável, não existindo qualquer tipo de escalabilidade do algoritmo. O algoritmo não escalou, apresentando tempos de execução superiores à medida que o número de *threads* é incrementado.

A paralelização deste algoritmo apresenta esta $performance$ pois apesar da carga computacional ser distribuída por diferentes *threads*, pelos diferentes cores, temos o fator da computação concorrente que impõe muita sincronização neste algoritmo o que o acaba por atrasar, e ainda haver o fator das *threads* correrem sequencialmente pela *heap*. Dado isto, temos ainda a múltipla utilização massiva da cache pelas diferentes *threads*, que acedem a uma estrutura de dados que é contínua na memória, mas o acesso aos seus elementos é muito irregular, existindo os saltos na verificação dos elementos filhos do nó em questão. Tudo isto leva a que cada *thread* manche a *cache* ainda utilizada por outros fio de execução e pelo grande número de elementos, levando assim a um aumento do número de *cache-misses*, visível na Figura 6, afetando assim drasticamente o desempenho da $HeapSort$.

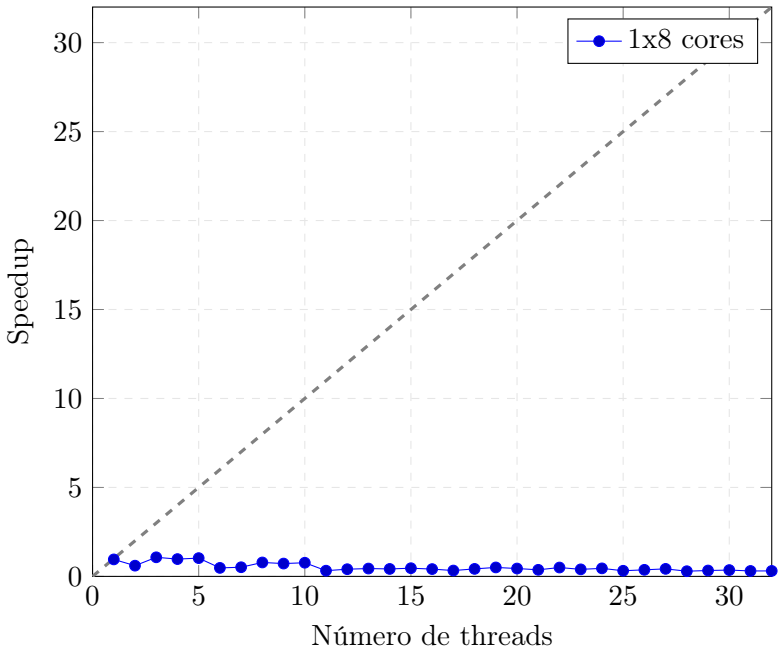

Figura 5: *Speedup* heapsort da versão concorrente em C++

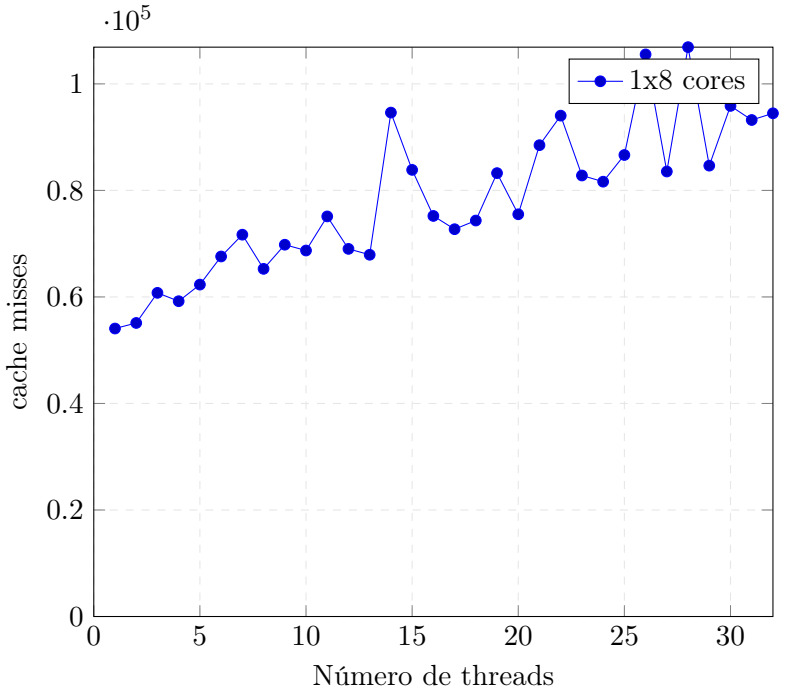

Figura 6: Número de *cache-misses* da heapsort em C++

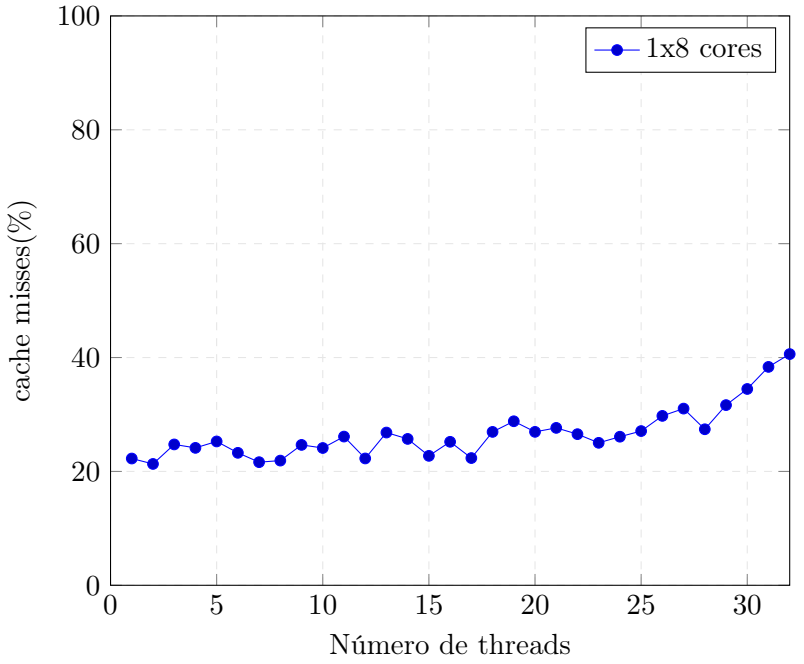

Figura 7: Percentagem de *cache-misses* da heapsort em C++

4 Conclusão

A implementação paralela deste algoritmo revelou um grau de dificuldade acima do esperado devido ao elevado número de casos em que é necessária a execução, em exclusão mútua, de secções de código de forma a não existirem *data races*. Para além disso as alternativas criadas tinham comportamentos diferentes quando executadas em máquinas ou sistemas operativos diferentes, isto é, por exemplo, em algumas máquinas o programa entrava em *deadlock* e em outras não, o que se deveu também ao facto de se utilizarem versões diferentes de compiladores. Este problema de ferramentas desatualizadas, no *cluster* deu-se uma vez mais com o uso do *cmake*, uma ferramenta para facilitar o processo de compilação da linguagem `C++`.

A implementação de diferentes alternativas de paralelização também teve os seus contratempos, por exemplo o código que utiliza *tasks* foi inicialmente escrito em `C` com recurso ao *openMP*, num segunda fase foi reescrito em `C++` com recurso á biblioteca *Boost*, mas esta biblioteca não estava atualizada no *cluster*, e portanto não tinha as funcionalidades necessárias, por fim foi utilizada a linguagem *Java* que facilitou o processo de compilação assim como permitiu o uso de uma *thread pool* que facilitou a implementação do algoritmo.

A deteção de erros destas implementações revelaram ser uma tarefa difícil dada à forte vertente de programação concorrente utilizada, e dado aos mecanismos utilizados para gerir os acessos de recursos partilhados e para conter os *data races*. Pois a deteção de ordenação correta dos elementos nem sempre era realizada, mesmo parecendo que este processo era realizado com sucesso. A realização do *debug* do código também incorpora este mesmo fenómeno, e dada que a utilização de ferramentas de *debug* para a aplicações que utilizam diversos fios de execução trazem uma dificuldade muito mais acrescida.

Por fim houve também a dificuldade na realização das medições, pois apenas um nodo do *cluster* tinha a ferramenta *perf* funcional, o que levou a que as medições tivessem de ser feitas numa maquina pessoal. Esta última fase não foi possível ser feita com o detalhe desejado devido à falta de tempo a que todos estes problemas conduziram.

5 Anexos

5.1 Característica do computador pessoal *Quad-core*

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	8
On-line CPU(s) list	0-7
Thread(s) per core	2
Core(s) per socket	4
CPU socket(s)	1
NUMA node(s)	1
Vendor ID	GenuineIntel
CPU family	6
Model	158
Model name	Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Stepping:	9
CPU MHz:	1090.124
CPU max MHz:	3800,0000
CPU min MHz:	800,0000
BogoMIPS:	5616.00
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	6144K
NUMA node0 CPU(s):	0-7

Tabela 1: Especificação de MSI GL62M 7RD